InterSystems SQL Reference

Version 2023.1
2024-07-11

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)

Tel:	+1-617-621-0700
Tel:	+44 (0) 844 854 2917
Email:	support@InterSystems.com

Table of Contents

List of Tables

Symbols and Syntax Conventions

Symbols Used in InterSystems SQL

A table of characters used in InterSystems SQL as operators, etc.

Table of Symbols

The following are the literal symbols used in InterSystems SQL on InterSystems IRIS® data platform. (This list does not include symbols indicating format conventions, which are not part of the language.) There is a separate table for symbols used in ObjectScript.

The name of each symbol is followed by its ASCII decimal code value.

Symbol	Name and Usage
[space] or [tab]	*White space (Tab (9) or Space (32)):* One or more whitespace characters between keywords, identifiers, and variables.
!	*Exclamation mark (33):* OR logical operator in between predicates in condition expressions. Used in the WHERE clause, the HAVING clause, and elsewhere. In SQL Shell, the ! command is used to issue an ObjectScript command line.
!=	*Exclamation mark/Equal sign:* Is not equal to comparison condition.
"	*Quotes (34):* Encloses a delimited identifier name. In Dynamic SQL used to enclose literal values for class method arguments, such as SQL code as a string argument for the %Prepare() method, or input parameters as string arguments for the %Execute() method. In %PATTERN used to enclose a literal value within a pattern string. For example, `'3L1"L".L'` (meaning 3 lowercase letters, followed by the capital letter "L", followed by any number of lowercase letters). In XMLELEMENT used to enclose a tag name string literal.
""	*Two quotes:* By themselves, an invalid delimited identifier. Within a delimited identifier, an escape sequence for a literal quote character. For example, `"a""good""id"`.
#	*Pound sign (35):* Valid identifier name character (not first character). With spaces before and after, modulo arithmetic operator. For Embedded SQL, ObjectScript macro preprocessor directive prefix. For example, `#include`. In SQL Shell the # command is used to recall statements from the SQL Shell history buffer.
$	*Dollar sign (36):* Valid identifier name character (not first character). First character of some InterSystems IRIS extension SQL functions.
$$	*Double dollar sign:* used to call an ObjectScript user-defined function (also known as an extrinsic function). For more details, see Function and Method Call Selection in the *selectItem* argument of the **SELECT** reference page.

Symbol	Name and Usage
%	*Percent sign (37):* Valid first character for identifier names (first character only). First character of some InterSystems SQL extensions to the SQL standard, including string collation functions (%SQLUPPER), aggregate functions (%DLIST), and predicate conditions (%STARTSWITH). First character of %ID, %TABLENAME, and %CLASSNAME keywords in SELECT. First character of some privilege keywords (%CREATE_TABLE, %ALTER) and some role names (%All). First character of some Embedded SQL system variables (%ROWCOUNT, %msg). Data type max length indicator: CHAR(%24) LIKE condition predicate multi-character wildcard.
%%	*Double percent sign:* Prefix for the pseudo-field reference variable keywords: %%CLASS-NAME, %%CLASSNAMEQ, %%ID, and %%TABLENAME, used in ObjectScript computed field code and trigger code.
&	*Ampersand (38):* AND logical operator in WHERE clause and other condition expressions. $BITLOGIC bitstring And operator. Embedded SQL invocation prefix: &sql(*SQL commands*).
'	*Single quote character (39):* Encloses a string literal.
''	*Double single quote characters:* An empty string literal. An escape sequence for a literal single quote character within a string value. For example: `'can''t'`
( )	*Parentheses (40,41):* Encloses comma-separated lists. Encloses argument(s) of an SQL function. Encloses the parameter list for a procedure, method, or query. In most cases, the parentheses must be specified, even if no arguments or parameters are supplied. In a **SELECT** DISTINCT BY clause, encloses an item or comma-separated list of items used to select unique values. In a **SELECT** statement, encloses a subquery in the FROM clause. Encloses the name of a predefined query used in a UNION. Encloses host variable array subscripts. For example, `INTO :var(1),:var(2)` Encloses embedded SQL code: &sql(code) Used to enforce precedence in arithmetic operations: 3+(3*5)=18. Used to group predicates: WHERE NOT (Age<20 AND Age>12).
(( ))	*Double Parentheses:* suppress literal substitution in cached queries. For example, SELECT TOP ((4)) Name FROM Sample.Person WHERE Name %STARTSWITH (('A')). Optimizes WHERE clause selection of a non-null outlier value.

Symbol	Name and Usage
*	*Asterisk (42):* A wildcard, indicating "all" in the following cases: In SELECT retrieve all columns: SELECT * FROM table. In COUNT, count all rows (including nulls and duplicates). In GRANT and REVOKE, all basic privileges, all tables, or all currently defined users. In %MATCHES pattern string a multi-character wildcard. Multiplication arithmetic operator.
*/	*Asterisk slash:* Multi-line comment ending indicator. Comment begins with /*.
+	*Plus sign (43):* Addition arithmetic operator. Unary positive sign operator.
,	*Comma (44):* List separator, for example, multiple field names. In data size definition: NUMERIC (precision,scale).
–	*Hyphen (minus sign) (45):* Subtraction arithmetic operator. Unary negative sign operator. SQLCODE error code prefix: –304. Date delimiter. In %MATCHES pattern string a range indicator specified within square brackets. For example, `[a-m]`.
——	*Double hyphen:* Single-line comment indicator.
–>	*Hyphen, greater than (arrow):* implicit join arrow syntax.
.	*Period (46):* Used to separate parts of multipart names, such as qualified table names: schema.tablename, or column names: tablealias.fieldname Decimal point for numeric literals in American numeric format. Date delimiter for Russian, Ukrainian, and Czech locales: DD.MM.YYYY Prefixed to a variable or array name, specifies passing by reference: `.name` %PATTERN pattern string multi-character wildcard.
/	*Slash (47):* Division arithmetic operator. Date delimiter.
/*	*Slash asterisk:* Multi-line comment begins indicator. Comment ends with */.
:	*Colon (58):* Host variable indicator prefix: `:var` A time delimiter for hours, minutes, and seconds. In **CAST** and **CONVERT** functions, an optional thousandth-of-a-second delimiter. In trigger code a prefix indicating an ObjectScript label line. In **CREATE PROCEDURE** ObjectScript code body, a macro preprocessor directive prefix. For example, `:#include`.
::	*Double colon:* In trigger code this doubled prefix indicates that the identifier (::*name*) beginning that line is a host variable, not a label line.

Symbol	Name and Usage
;	*Semicolon (59):* SQL end of statement delimiter in procedures, methods, queries, and trigger code. Accepted as an optional end of statement delimiter by ImportDDL() or wherever specifying SQL code using a TSQL dialect. Otherwise, InterSystems SQL does not use or allow a semicolon at the end of an SQL statement.
<	*Less than (60):* Less than comparison condition.
<=	*Less than or equal to:* Less than or equal to comparison condition.
<>	*Less than/Greater than:* Is not equal to comparison condition.
=	*Equal sign (61):* Equal to comparison condition. In WHERE clause, an Inner Join.
>	*Greater than (62):* Greater than comparison condition.
>=	*Greater than or equal to:* Greater than or equal to comparison condition.
?	*Question mark (63):* In Dynamic SQL, an input parameter variable supplied by the Execute method. In %MATCHES pattern string a single-character wildcard. In SQL Shell the ? command displays help text for SQL Shell commands.
@	*At sign (64):* Valid identifier name character (not first character).
E, e	*The letter "E" (69, 101):* Exponent indicator. %PATTERN code specifying any printable character.
[	*Open square bracket (91):* Contains predicate. Used in the WHERE clause, the HAVING clause, and elsewhere.
[ ]	*Open and close square brackets:* In %MATCHES pattern string, encloses a list or range of match characters. For example, `[abc]` or `[a-m]`.
\	*Backslash (92):* Integer division arithmetic operator. In %MATCHES pattern string an escape character.
]	*Close square bracket (93):* Follows predicate. Used in the WHERE clause, the HAVING clause, and elsewhere.
^	*Caret (94):* In %MATCHES pattern string a NOT character. For example, `[^abc]`.
_	*Underscore (95):* Valid first (or subsequent) character for identifier names. Valid first character for certain user names (but not passwords). Used in column names to represent embedded serial class data: SELECT Home_State, where Home is a field that references a serial class and State is a property defined in that serial class. LIKE condition predicate single-character wildcard.
{}	*Curly braces (123,125):* Enclose ODBC scalar functions: `{fn name(...)}`. Enclose time and date construct functions: `{d 'string'}`, `{t 'string'}`, `{ts 'string'}`. Enclose ObjectScript code in procedures, methods, queries, and trigger code.

Symbol	Name and Usage
||	*Double vertical bar (124):* Concatenation operator.
	Compound ID indicator. Used by InterSystems IRIS as a delimiter between multiple properties in a generated compound object ID (a concatenated ID). This can be either an IDKey index defined on multiple properties (`prop1||prop2`), or an ID for a parent/child relationship (`parent||child`). Cannot be used in IDKEY field data.

Syntax Conventions

Specifies conventions used in the InterSystems SQL Reference.

Description

The following are the format conventions used in this reference. These format characters explain usage; they are *not* specified when coding an SQL program. For a table of the symbols that are used in SQL coding, refer to the SQL Symbols table.

Symbol	Meaning
[*nnnn*]	An argument enclosed in square brackets is optional. Specify none or one.
{ *nnnn* }	An argument enclosed in curly braces is optional, and may be repeated multiple times. Specify none, one, or more than one. Curly braces are also used as literal characters, for example in ODBC scalar functions with the form: {fn FUNCTION(*arg*)}
mmmm | *nnnn*	A vertical bar means OR. Specify either one or the other.
. . .	An ellipsis indicates an unspecified portion of a complete SQL statement. It can also be used to specify repetition: *var1*,*var2*,...
::=	Is equivalent to.

If an argument appears as an "*item-list*", then the argument can consist of one or more of the particular *item*s delimited by a particular character. A cross-reference from an *item-list* points to the page for *item* itself.

If an argument appears as an "*item-commalist*", then the argument can consist of one or more of the particular *item*s delimited by a comma. A cross-reference from an *item-commalist* points to the page for *item* itself.

When an item is listed in bracketed parentheses, such as `[(]` `identifier` `[)]` then the *pair* of parentheses (as a unit) is optional.

SQL Commands

ALTER FOREIGN SERVER (SQL)

Alters a foreign server definition.

Synopsis

Change Connection

```
ALTER [ FOREIGN ] SERVER server-name ALTER CONNECTION jdbc-connection
ALTER [ FOREIGN ] SERVER server-name ALTER HOST file-path
```

Change Delimited Identifiers

```
ALTER [ FOREIGN ] SERVER server-name ALTER id-option
```

Change Connection and Delimited Identifiers

```
ALTER [ FOREIGN ] SERVER server-name MODIFY [ CONNECTION jdbc-connection | HOST file-path ], id-option
```

Arguments

Arguments	Description
server-name	The name for the foreign server definition being altered. A valid identifier.
CONNECTION cxn-name	The name of the new JDBC connection that will connect InterSystems IRIS with an external data source. A valid identifier. Must be the name of a JDBC connection that has already been defined. Must be delimited.
HOST file-path	The new file path that you want to use to access files that will be projected into InterSystems IRIS.
id-option	Either DELIMITEDIDS or NODELIMITEDIDS. Sets behavior based on whether the external data source accepts delimited identifiers or not.

Description

The ALTER FOREIGN SERVER command allows you to change how a foreign server connects with an external data source. You may use the ALTER variant of the command to change a single parameter or the MODIFY variant to change multiple parameters. In particular, you may change the file path, JDBC connection, or delimited identifier option that the foreign server uses when connecting with an external source.

Before you change the connection parameters of a foreign server with either the CONNECTION or HOST property, you should be sure that your changes will not affect your ability to access the foreign tables you have defined on the foreign server. For example, if you change the HOST file path and still want to access the tables you have already defined, you should move any .csv files associated with foreign tables into the new file path. You will be unable to access data in these tables until you have made the proper changes. There are no concerns when you use ALTER FOREIGN SERVER to change these connection parameters on a foreign server that does not have a foreign table defined on it.

Examples

The following example alters a foreign server's file path to read data from a different directory.

```
ALTER FOREIGN SERVER Sample.Test ALTER HOST '/second/filepath'
```

The following example alters a foreign server's JDBC connection to read data from a different database source and indicates that the external data source permits delimited identifiers.

```
ALTER FOREIGN SERVER Sample.Test MODIFY CONNECTION 'anotherConnection', DELIMITEDIDS
```

See Also

- CREATE FOREIGN SERVER

- DROP FOREIGN SERVER

- CREATE FOREIGN TABLE

ALTER FOREIGN TABLE (SQL)

Alters a foreign table definition.

Synopsis

Change Column Name

```
ALTER FOREIGN TABLE table-name ALTER [ COLUMN ] old-name RENAME new-name
ALTER FOREIGN TABLE table-name MODIFY old-name RENAME new-name, old-name2 RENAME new-name2, ...
ALTER FOREIGN TABLE table-name ALTER [ COLUMN ] old-name RENAME new-name VALUES ( external-name )
ALTER FOREIGN TABLE table-name MODIFY old-name RENAME new-name, old-name2 RENAME new-name2, ... VALUES
 ( newexternal-name, newexternal-name2, ... )
```

Change Datatypes

```
ALTER FOREIGN TABLE table-name ALTER col-name datatype
ALTER FOREIGN TABLE table-name MODIFY col-name datatype {, col-name datatype ...}
```

Arguments

Arguments	Description
table-name	The name for the foreign table that will be altered. A valid identifier. Must be the name of a foreign table that exists on a foreign server before this command is issued.
old-name	The name of the column within InterSystems IRIS that will be changed. A valid identifier. Must correspond with the name of a column that exists in the foreign table before this command is issued.
new-name	The new name of the column within InterSystems IRIS. A valid identifier.
external-name	The new name of the column in the external data source that projects data into the corresponding column in the RENAME clause.
col-name	The name of the column that will be converted to a new data type. A valid identifier. Must correspond with the name of a column that exists in the foreign table before this command is issued.
dataype	The new datatype of the column. Must be a valid SQL data type.

Description

The ALTER FOREIGN TABLE command modifies a foreign table definition. There are two types of alterations you may apply to a given table:

- Change the column name(s) of a column or list of columns.

- Change the data type(s) of a column or list of columns.

Change Column Names

You may use the ALTER FOREIGN TABLE command to change the column names of a single column or a list of columns in a foreign table.

There are two variations:

- **ALTER FOREIGN TABLE** *table-name* **ALTER [COLUMN]** *old-name* **RENAME** *new-name* renames a column of the foreign table from *old-name* to *new-name*.

- **ALTER FOREIGN TABLE** *table-name* **MODIFY** *old-name* **RENAME** *new-name* renames one or more columns of the foreign table from their *old-name* to their corresponding *new-name*.

- **ALTER FOREIGN TABLE** *table-name* **ALTER [COLUMN]** *old-name* **RENAME** *new-name* **VALUES (** *external-name* **)** renames a column of the foreign table from *old-name* to *new-name*. This variation also changes the name of the column in the external data source that projects data into the specified column.

- **ALTER FOREIGN TABLE** *table-name* **MODIFY** *old-name* **RENAME** *new-name***,** *old-name2* **RENAME** *new-name2* **VALUES (** *external-name***,** *external-name2* **)** renames a series of columns of the foreign table from the *old-name* to the corresponding *new-name*. This variation also changes the names of the columns in the external data source that projects data into the specified columns.

Note: InterSystems does not recommend changing the name of a column or set of columns with the ALTER FOREIGN TABLE command. Instead, because a foreign table is merely a projection of data from another source, if you intend to make significant changes to the external data source, you should drop the foreign table, edit the database or .csv file, and then recreate the foreign table.

Change Column Data Types

You may use the ALTER FOREIGN TABLE command to convert the data types of a column or a list of columns in a foreign table. The new data type(s) must be valid InterSystems SQL data type(s).

You may not change the data type of a column if the change would result in stream data being typed as non-stream data or non-stream data being typed as stream data. Attempting to do so results in a SQLCODE -374 error.

There are two variations:

- **ALTER FOREIGN TABLE** *table-name* **ALTER** *col-name datatype* changes the data type of a single column.

- **ALTER FOREIGN TABLE** *table-name* **MODIFY** *col-name datatype* **{,** *col-name datatype* **...}** changes the data type(s) of one or more columns. You may specify a different data type for each column.

Examples

The following example change the names of the LastName column on a foreign table called Sample.Person. The example shows both the ALTER and MODIFY forms of the command.

```
ALTER FOREIGN TABLE Sample.Person ALTER COLUMN LastName RENAME Surname
ALTER FOREIGN TABLE Sample.Person MODIFY LastName RENAME FamilyName, FirstName RENAME GivenName
```

The following example changes the data type of the Amount column on a foreign table called Sample.Account. The example shows both the ALTER and MODIFY forms of the commands.

```
ALTER FOREIGN TABLE Sample.Person ALTER Amount INTEGER
ALTER FOREIGN TABLE Sample.Person MODIFY Amount INTEGER
```

See Also

- CREATE FOREIGN TABLE
- DROP FOREIGN TABLE

ALTER ML CONFIGURATION (SQL)

Modifies an ML configuration.

Synopsis

```
ALTER ML CONFIGURATION ml-configuration-name
    [ PROVIDER provider-name]
    [ %DESCRIPTION description]
    [ USING json-object-string ]
    [ provider-connection-settings ]
```

Arguments

ml-configuration-name	The name for the ML configuration being altered.
PROVIDER *provider-name*	A string specifying the name of a machine learning provider, where values are: • AutoML • H2O • DataRobot • PMML
%DESCRIPTION *description*	*Optional* — String. A text description for the ML configuration. See details below.
USING *json-object-string*	*Optional* — A JSON string specifying one or more key-value pairs; see details below.
provider-connection-settings	Any additional settings, required for connection, that vary by the machine learning provider. See details below.

Description

The **ALTER ML CONFIGURATION** statement alters one, or several, parameters within an ML configuration definition. You can alter:

• The provider

• The description

• The USING clause

• Provider connection settings

ML Configuration Description

%DESCRIPTION accepts a text string enclosed in single quotes, which you can use to provide a description for documenting your configuration. This text can be of any length, and can contain any characters, including blank spaces.

USING

You can specify a default **USING** clause for your configuration. This clause accepts a JSON string with one or more key-value pairs. When **TRAIN MODEL** is executed, by default the **USING** clause of the configuration is used.

```
ALTER ML CONFIGURATION MyConfiguration USING {"seed": 3}
```

You must make sure that the parameters you specify are recognized by the provider you select. Failing to do so may result in an error when training.

Provider Connection Settings

Depending on the provider specified by your configuration, there may be additional fields you must enter to establish a successful connection.

DataRobot

You must specify the following values to successfully connect to DataRobot:

- `URL [ = ] url-string` — where `url-string` is the URL of a DataRobot endpoint.

- `APITOKEN [ = ] token-string` — where `token-string` is your client API token to access the DataRobot AutoML server.

Altering an ML configuration for DataRobot could be performed with a query as follows:

```
ALTER ML CONFIGURATION datarobot-configuration URL url-string APITOKEN token-string
```

With proper values for `url-string` and `token-string`

Required Security Privileges

Calling **ALTER ML CONFIGURATION** requires %ALTER_ML_CONFIGURATION privileges; otherwise, there is a SQLCODE –99 error (Privilege Violation). To assign %ALTER_ML_CONFIGURATION privileges, use the GRANT command.

Examples

The following SQL query edits an existing configuration named `TestH2O` to add a **USING** clause that the user wants used for every model being trained:

```
ALTER ML CONFIGURATION TestH2O USING {"seed": 2}
```

See Also

- CREATE ML CONFIGURATION, DROP ML CONFIGURATION

ALTER MODEL (SQL)

Modifies a model

Synopsis

```
ALTER MODEL model-name PURGE [ ALL ] [ integer DAYS ]
```

```
ALTER MODEL model-name DEFAULT [ TRAINED MODEL ] trained-model-name
```

Arguments

model-name	The name of the machine learning model to alter.
DEFAULT *trained-model-name*	A trained machine learning model.
integer DAYS	An integer.

Description

An **ALTER MODEL** statement modifies a machine learning model. You can perform only one type of operation in each ALTER MODEL statement.

- A PURGE deletes all training runs and validation runs for the associated model based on the given scope:

    - If no scope is given, all records are deleted except for those associated with the default trained model.

    - If `integer DAYS` is given, all records older than `integer` days are deleted.

    - If `ALL` is given, all records are deleted regardless of when they occurred.

- A DEFAULT (or DEFAULT TRAINED MODEL) sets the default trained model to be the model specified. This is useful when you have made several **TRAIN MODEL** statements using the same model definition, saving each trained model to a different name, and you wish to switch which model the default name points to. Specifying a nonexistent model results in an error.

Required Security Privileges

Calling **ALTER MODEL** requires %MANAGE_MODEL privileges; otherwise, there is a SQLCODE –99 error (Privilege Violation). To assign %MANAGE_MODEL privileges, use the GRANT command.

Examples

The following query uses a PURGE clause to delete all training and validation run data for the `SpamFilter` model:

```
ALTER MODEL SpamFilter PURGE ALL
```

The following query uses a DEFAULT clause to change the default trained model of `SpamFilter` to `SpamFilter3`

```
ALTER MODEL SpamFilter DEFAULT SpamFilter3
```

See Also

- CREATE MODEL, DROP MODEL

ALTER TABLE (SQL)

Modifies a table.

Synopsis

```
ALTER TABLE table alter-action

where alter-action is one of the following:
    ADD  [(] add-action {,add-action} [)] |
    DROP [COLUMN ] drop-column-action {,drop-column-action} |
    DROP drop-action |
    DELETE drop-action |
    ALTER [COLUMN] field alter-column-action |
    MODIFY modification-spec {,modification-spec}
    RENAME table

add-action ::=
    [CONSTRAINT identifier]
    [(] FOREIGN KEY (field-commalist)
        REFERENCES table (field-commalist)
        [ON DELETE ref-action] [ON UPDATE ref-action]
        [NOCHECK]  [)]
    |
    [(] UNIQUE (field-commalist)  [)]
    |
    [(] PRIMARY KEY (field-commalist) [)]
    |
    DEFAULT [(] default-spec [)] FOR field
    |
    [COLUMN] [(] field datatype  [sqlcollation]
        [%DESCRIPTION string]
        [DEFAULT [(] default-spec [)] ]
        [ON UPDATE update-spec ]
        [UNIQUE] [NOT NULL]
        [REFERENCES table (field-commalist)
        [ON DELETE ref-action] [ON UPDATE ref-action]
        [NOCHECK]  ]
    [)]

drop-column-action ::=
        [COLUMN] field [RESTRICT | CASCADE] [%DELDATA | %NODELDATA]

drop-action ::=
    FOREIGN KEY identifier |
    PRIMARY KEY |
    CONSTRAINT identifier |

alter-column-action ::=
    RENAME newfieldname |
    datatype |
    [SET] DEFAULT [(] default-spec [)] | DROP DEFAULT |
    NULL | NOT NULL |
    COLLATE sqlcollation

modification-spec ::=
    oldfieldname RENAME newfieldname |
    field [datatype]
        [DEFAULT [(] default-spec [)]]
        [CONSTRAINT identifier] [NULL] [NOT NULL]

sqlcollation ::=
    { %EXACT | %MINUS | %MVR | %PLUS | %SPACE |
        %SQLSTRING [(maxlen)] | %SQLUPPER [(maxlen)] |
        %TRUNCATE[(maxlen)]  }
```

Arguments

Argument	Description
table	The name of the table to be altered. The table name can be qualified (schema.table), or unqualified (table). An unqualified table name takes the default schema name. Schema search path values are not used.
identifier	A unique name assigned to a constraint. Must be a valid identifier.
field	The name of the column to be altered (added, modified, deleted). Must be a valid identifier.
field-commalist	The name of a column or a comma-separated list of columns. An *field-commalist* must be enclosed in parentheses, even when only a single column is specified. See SQL Identifiers.
datatype	A valid InterSystems SQL data type. See Data Types.
default-spec	A default data value automatically supplied for this field, if not overridden by a user-supplied data value. Allowed values are: a literal value; one of the following keyword options (NULL, USER, CURRENT_USER, SESSION_USER, SYSTEM_USER, CURRENT_DATE, CURRENT_TIME, and CURRENT_TIMESTAMP); or an OBJECTSCRIPT expression. Do not use the SQL zero-length string as a default value. For further details, see CREATE TABLE.
update-spec	See ON UPDATE in **CREATE TABLE**.
COLLATE *sqlcollation*	*Optional* — Specify one of the following SQL collation types: %EXACT, %MINUS, %PLUS, %SPACE, %SQLSTRING, %SQLUPPER, %TRUNCATE, or %MVR. The default is the namespace default collation (%SQLUPPER, unless changed). %SQLSTRING, %SQLUPPER, and %TRUNCATE may be specified with an optional maximum length truncation argument, an integer enclosed in parentheses. The percent sign (%) prefix to these collation parameter keywords is optional. The COLLATE keyword is optional. For further details refer to Table Field/Property Definition Collation.

Description

An **ALTER TABLE** statement modifies a table definition; it can add elements, remove elements, or modify existing elements. You can only perform one type of operation in each **ALTER TABLE** statement.

- RENAME can rename a table, or can rename an existing column in a table with either ALTER COLUMN or MODIFY syntax.

- ADD can add multiple columns and/or constraints to a table. You specify the ADD keyword once, followed by a comma-separated list. You can use a comma-separated list to add multiple new columns to a table, add a list of constraints to existing columns, or both add new columns and add constraints to existing columns.

- DROP COLUMN can delete multiple columns from a table. You specify the DROP keyword once, followed by a comma-separated list of columns each with their optional cascade and/or data-delete option.

- ALTER COLUMN can change the definition of a single column. It cannot alter multiple columns.

- MODIFY can change the definition of a single column or a comma-separated list of columns. It does not support all of the options provided by ALTER COLUMN.

- DROP can drop a constraint from a field or group of fields. DROP can only operate on a single constraint.

The **ALTER TABLE** DROP keyword and the **ALTER TABLE** DELETE keyword are synonyms.

To determine if a specified table exists in the current namespace, use the **$SYSTEM.SQL.Schema.TableExists()** method.

Privileges and Locking

The **ALTER TABLE** command is a privileged operation. The user must have %ALTER_TABLE administrative privilege to execute **ALTER TABLE**. Failing to do so results in an SQLCODE –99 error with the %msg `User 'name' does not have %ALTER_TABLE privileges.`

The user must have %ALTER privilege on the specified table. If the user is the Owner (creator) of the table, the user is automatically granted %ALTER privilege for that table. Otherwise, the user must be granted %ALTER privilege for the table. Failing to do so results in an SQLCODE –99 error with the %msg `User 'name' does not have required %ALTER privilege needed to change the table definition for 'Schema.TableName'.`

To determine if the current user has %ALTER privilege, invoke the %CHECKPRIV command. To determine if a specified user has %ALTER privilege, invoke the **$SYSTEM.SQL.Security.CheckPrivilege()** method.

To assign the required administrative privilege, use the **GRANT** command with %ALTER_TABLE privilege; this requires the appropriate granting privileges. To assign the %ALTER object privilege, you can use:

- The **GRANT** command with the %ALTER privilege. This requires the appropriate granting privileges.

- The **ALTER** check box for the table on the **SQL Tables** tab in the Management Portal on the page for editing a role or user. This requires the appropriate granting privileges.

In embedded SQL, you can use the **$SYSTEM.Security.Login()** method to log in as a user with appropriate privileges:

ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql(      )
```

You must have the **%Service_Login:Use** privilege to invoke the **$SYSTEM.Security.Login** method. For further information, see %SYSTEM.Security.

- **ALTER TABLE** cannot be used on a table projected from a persistent class, unless the table class definition includes [DdlAllowed]. Otherwise, the operation fails with an SQLCODE -300 error with the %msg `DDL not enabled for class 'Schema.tablename'.`

- **ALTER TABLE** cannot be used on a table projected from a deployed persistent class. This operation fails with an SQLCODE -400 error with the %msg `Unable to execute DDL that modifies a deployed class: 'classname'.`

ALTER TABLE acquires a table-level lock on *table*. This prevents other processes from modifying the table's data. This lock is automatically released at the conclusion of the **ALTER TABLE** operation. When **ALTER TABLE** locks the corresponding class definition, it uses the SQL Lock Timeout setting for the current process.

To alter a table, the table cannot be locked by another process in either EXCLUSIVE MODE or SHARE MODE. Attempting to alter a locked table results in an SQLCODE -110 error, with a %msg such as the following: `Unable to acquire exclusive table lock for table 'Sample.MyTest'.`

RENAME Table

You can rename an existing table using the following syntax:

```
ALTER TABLE schema.TableName RENAME NewTableName
```

This operation renames the existing table in its existing schema. You can only change the table name, not the table schema. Specifying a schema name in the NewTableName results in an SQLCODE -1 error. Specifying the same table name for both old and new tables generates an SQLCODE -201 error.

Renaming a table changes the SQL table name. It does not change the corresponding persistent class name.

Renaming a table does not change references to the old table name in triggers.

If a view references the existing table name, attempting to rename the table will fail. This is because attempting to rename the table is an atomic operation that causes a recompile of the view, which generates an SQLCODE -30 error "`Table 'schema.oldname' not found`".

ADD COLUMN Restrictions

ADD COLUMN can add a single column, or can add a comma-separated list of columns.

If you attempt to add a field to a table through an **ALTER TABLE** *tablename* ADD COLUMN statement:

- If a column of that name already exists, the statement fails with an SQLCODE -306 error.

- If the statement specifies a NOT NULL constraint on the column *and* there is no default value for the column, then the statement fails if data already exists in the table. This is because, after the completion of the DDL statement, the NOT NULL constraint is not satisfied for all the pre-existing rows. This generates the error code SQLCODE -304 (Attempt to add a NOT NULL field with no default value to a table which contains data).

- If the statement specifies a NOT NULL constraint on the column *and* there is a default value for the column, the statement updates any existing rows in the table and assigns the default value for the column to the field. This includes default values such as CURRENT_TIMESTAMP.

- If the statement DOES NOT specify a NOT NULL constraint on the column *and* there is a default value for the column, then there are no updates of the column in any existing rows. The column value is NULL for those rows.

To change this default NOT NULL constraint behaviors, refer to the COMPILEMODE=NOCHECK option of the SET OPTION command.

If you specify an ordinary data field named "ID" and the RowID field is already named "ID" (the default), the ADD COLUMN operation succeeds. **ALTER TABLE** adds the ID data column, and renames the RowId column as "ID1" to avoid duplicate names.

Adding an Integer Counter

If you attempt to add an integer counter field to a table through an **ALTER TABLE** *tablename* ADD COLUMN statement:

- You can add an IDENTITY field to a table if the table does not have an IDENTITY field. If the table already has an IDENTITY field, the ALTER TABLE operation fails with an SQLCODE -400 error with a %msg such as the following: `ERROR #5281: Class has multiple identity properties: 'Sample.MyTest::MyIdent2'`. When you use ADD COLUMN to define this field, InterSystems IRIS populates existing data rows for this field using the corresponding RowID integer values.

  If **CREATE TABLE** defined a bitmap extent index and later you add an IDENTITY field to the table, and the IDENTITY field is not of type %BigInt, %Integer, %SmallInt, or %TinyInt with a MINVAL of 1 or higher, and there is no data in the table, the system automatically drops the bitmap extent index.

- You can add one or more SERIAL (%Library.Counter) fields to a table. When you use ADD COLUMN to define this field, existing data rows are NULL for this field. You can use **UPDATE** to supply values to existing data rows that are NULL for this field; you cannot use **UPDATE** to change non-NULL values.

- You can add a ROWVERSION field to a table if the table does not have a ROWVERSION field. If the table already has a ROWVERSION field, the ALTER TABLE operation fails with an SQLCODE -400 error with a %msg such as the following: `ERROR #5320: Class 'Sample.MyTest' has more than one property of type %Library.RowVersion. Only one is allowed. Properties: MyVer,MyVer2`. When you use ADD COLUMN to define this field, existing data rows are NULL for this field; you cannot update ROWVERSION values that are NULL.

ALTER COLUMN Restrictions

ALTER COLUMN can modify the definition of a single column:

- Rename the column using the syntax `ALTER TABLE tablename ALTER COLUMN oldname RENAME newname`. Renaming a column changes the SQL field name. It does not change the corresponding persistent class property name. ALTER COLUMN oldname RENAME newname replaces oldfield name references in trigger code and ComputeCode.

- Change the column characteristics: data type, default value, NULL/NOT NULL, and collation type.

If the table contains data, you cannot change the data type of a column that contains data if this change would result in stream data being typed as non-stream data or non-stream data being typed as stream data. Attempting to do so results in an SQLCODE -374 error. If there is no existing data, this type of datatype change is permitted.

You can use ALTER COLUMN to add, change, or drop a field default value.

If the table contains data, you cannot specify NOT NULL for a column if that column contains null values; this results in an SQLCODE -305 error.

If you change the collation type for a column that contains data, you must rebuild all indexes for that column.

MODIFY column Restrictions

MODIFY can modify the definitions of a single column or a comma-separated list of columns.

- Rename the column using the syntax `ALTER TABLE tablename MODIFY oldname RENAME newname`. Renaming a column changes the SQL field name. It does not change the corresponding persistent class property name. MODIFY oldname RENAME newname replaces oldfield name references in trigger code and ComputeCode.

- Change the column characteristics: data type, default value, and other characteristics.

If the table contains data, you cannot change the data type of a column that contains data to an incompatible data type:

- A data type with a lower (less inclusive) data type precedence if this conflicts with existing data values. Attempting to do so results in an SQLCODE -104 error, with the %msg specifying which field and which data value caused the error.

- A data type with a smaller MAXLEN or a MAXVAL/MINVAL if this conflicts with existing data values. Attempting to do so results in an SQLCODE -104 error, with the %msg specifying which field and which data value caused the error.

- A data type change from a stream data type to a non-stream data type or a non-stream data type to a stream data type. Attempting to do so results in an SQLCODE -374 error. If there is no existing data, this type of datatype change is permitted.

You can use MODIFY to add or change a field default value. You cannot use MODIFY to drop a field default value.

If the table contains data, you cannot specify NOT NULL for a column if that column contains null values; this results in an SQLCODE -305 error. The syntax forms `ALTER TABLE mytable MODIFY field1 NOT NULL` and `ALTER TABLE mytable MODIFY field1 CONSTRAINT nevernull NOT NULL` perform the same operation. The optional CONSTRAINT *identifier* clause is a no-op provided for compatibility. InterSystems IRIS does not retain or use this field constraint name. Attempting to drop this field constraint by specifying this field constraint name results in an SQLCODE -315 error.

DROP COLUMN Restrictions

DROP COLUMN can delete multiple column definitions, specified as a comma-separated list. Each listed column name must be followed by its RESTRICT or CASCADE (if unspecified, the default is RESTRICT) and %DELDATA or %NODELDATE (if unspecified, the default is %NODELDATA) options.

By default, deleting a column definition does not delete any data that has been stored in that column from the data map. To delete both the column definition and the data, specify the %DELDATA option.

Deleting a column definition does not delete the corresponding column-level privileges. For example, the privilege granted to a user to insert, update, or delete data on that column. This has the following consequences:

- If a column is deleted, and then another column with the same name is added, users and roles will have the same privileges on the new column that they had on the old column.

- Once a column is deleted, it is not possible to revoke object privileges for that column.

For these reasons, it is generally recommended that you use the REVOKE command to revoke column-level privileges from a column before deleting the column definition.

RESTRICT keyword (or no keyword): You cannot drop a column if that column appears in an index, or is defined in a foreign key constraint or other unique constraint. Attempting a DROP COLUMN for that column fails with an SQLCODE -322 error. RESTRICT is the default. See DROP INDEX.

CASCADE keyword: If the column appears in an index, the index will be deleted. There may be multiple indexes. If the column appears in a foreign key, the foreign key will be deleted. There may be multiple foreign keys.

You cannot drop a column if that column is used in COMPUTECODE or in a COMPUTEONCHANGE clause. Attempting to do so results in an SQLCODE -400 error.

ADD CONSTRAINT Restrictions

You can add a constraint to a comma-separated list of fields. For example, you can add the UNIQUE (FName,SurName) constraint, which establishes a UNIQUE constraint on the combined values of the two fields FName and Surname. Similarly, you can add a primary key constraint or a foreign key constraint on a comma-separated list of fields.

A constraint can be named or unnamed. If unnamed, InterSystems SQL generates a constraint name using the table name. For example, `MYTABLE_Unique1` or `MYTABLE_PKEY1`.

The following example creates two unnamed constraints, adding both the unique constraint and the primary key constraint to comma-separated lists of fields:

SQL

```
ALTER TABLE SQLUser.MyStudents
    ADD UNIQUE (FName,SurName),PRIMARY KEY (Fname,Surname)
```

- A field must exist to be used in a constraint. Specifying a non-existent field generates an SQLCODE -31 error.

- The RowId field cannot be used in a constraint. Specifying the RowId (ID) field generates SQLCODE -31 error.

- A stream field cannot be used in a constraint. Specifying a stream field generates an SQLCODE -400 error: "invalid index attribute"

- A constraint can only be applied once to a field. Specifying the same constraint twice to a field generates an SQLCODE -400 error: "index name conflict".

By using the optional CONSTRAINT *identifier* keyword clause, you can create a named constraint. A named constraint must be a valid identifier; constraint names are not case-sensitive. This provides a name for the constraint for future use. This is shown in the following example:

SQL

```
ALTER TABLE SQLUser.MyStudents
    ADD CONSTRAINT UnqFullName UNIQUE (FName,SurName)
```

You can specify multiple constraints as a comma-separated list; the constraint name is applied to the first constraint, the other constraints receive default names.

A constraint name must be unique for the table. Specifying the same constraint name twice to a field generates an SQLCODE -400 error: "index name conflict".

ADD PRIMARY KEY Restrictions

A primary key value is required and unique. Therefore, adding a primary key constraint to an existing field or combination of fields makes each of these fields a required field. If you add a primary key constraint to a list of existing fields, the combined values of these fields must be unique. You cannot add a primary key constraint to an existing field if that field permits NULL values. You cannot add a primary key constraint to a field (or list of fields) if that field (or list of fields) contain non-unique values.

If you add a primary key constraint to an existing field, the field may also be automatically defined as an IDKey index. This depends on whether data is present and upon a configuration setting established in one of the following ways:

- The SQL SET OPTION PKEY_IS_IDKEY statement.

- The system-wide **$SYSTEM.SQL.Util.SetOption()** method configuration option DDLPKeyNotIDKey. To determine the current setting, call **$SYSTEM.SQL.CurrentSettings()** which displays Are primary keys created through DDL not ID keys; the default is 1.

- Go to the Management Portal, select **System Administration**, **Configuration**, **SQL and Object Settings**, **SQL**. View the current setting of **Define primary key as ID key for tables created via DDL**.

  - If the check box is not selected (the default), the Primary Key does not becomes the IDKey index in the class definition. Access to records using a primary key that is not the IDKEY is significantly less efficient; however, this type of primary key value can be modified.

  - If the check box is selected, when a Primary Key constraint is specified through DDL, and the field does not contain data, the primary key index is also defined as the IDKey index. If the field *does* contain data, the IDKey index is not defined. If the primary key is defined as the IDKey index, data access is more efficient, but a primary key value, once set, can never be modified.

If **CREATE TABLE** defined a bitmap extent index and later you use **ALTER TABLE** to add a primary key that is also the IDKey, the system automatically drops the bitmap extent index.

ADD PRIMARY KEY When Already Exists

You can only define one primary key. By default, InterSystems IRIS rejects an attempt to define a primary key when one already exists, or to define the same primary key twice, and issues an SQLCODE -307 error. The SQLCODE -307 error is issued even if the second definition of the primary key is identical to the first definition. To determine the current configuration, call **$SYSTEM.SQL.CurrentSettings()**, which displays a Allow create primary key through DDL when key exists setting. The default is 0 (No), which is the recommended configuration setting. If this option is set to 1 (Yes), an **ALTER TABLE** ADD PRIMARY KEY causes InterSystems IRIS to remove the primary key index from the class definition, and then recreates this index using the specified primary key field(s).

From the Management Portal, **System Administration**, **Configuration**, **SQL and Object Settings**, **SQL** you can set this option (and other similar create, alter, and drop options) system-wide by selecting the **Ignore redundant DDL statements** check box.

However, even if this option is set to allow the creation of a primary key when one already exists, you cannot recreate a primary key index if it is also the IDKEY index and the table contains data. Attempting to do so generates an SQLCODE -307 error.

ADD FOREIGN KEY Restrictions

For information on foreign keys, refer to Defining Foreign Keys and Foreign Key Referential Action Clause in the **CREATE TABLE** command, and to Using Foreign Keys.

By default, you cannot have two foreign keys with the same name. Attempting to do so generates an SQLCODE -311 error. To determine the current setting, call **$SYSTEM.SQL.CurrentSettings()**, which displays a `Allow DDL ADD foreign key constraint when foreign key exists` setting. The default is 0 (No), which is the recommended setting for this option. When 1 (Yes), you can add a foreign key through DDL even if one with the same name already exists.

From the Management Portal, **System Administration**, **Configuration**, **SQL and Object Settings**, **SQL** you can set this option (and other similar create, alter, and drop options) system-wide by selecting the **Ignore redundant DDL statements** check box.

Your table definition should not have two foreign keys with different names that reference the same *field-commalist* field(s) and perform contradictory referential actions. In accordance with the ANSI standard, InterSystems SQL does not issue an error if you define two foreign keys that perform contradictory referential actions on the same field (for example, ON DELETE CASCADE and ON DELETE SET NULL). Instead, InterSystems SQL issues an error when a **DELETE** or **UPDATE** operation encounters these contradictory foreign key definitions.

An ADD FOREIGN KEY that specifies a non-existent foreign key field generates an SQLCODE -31 error.

An ADD FOREIGN KEY that references a non-existent parent key table generates an SQLCODE -310 error. An ADD FOREIGN KEY that references a non-existent field in an existing parent key table generates an SQLCODE -316 error. If you do not specify a parent key field, it defaults to the ID field.

Before issuing an ADD FOREIGN KEY, the user must have REFERENCES privilege on the table being referenced or on the columns of the table being referenced. REFERENCES privilege is required if the **ALTER TABLE** is executed via Dynamic SQL or over a SQL driver connection.

An ADD FOREIGN KEY that references a field (or combination of fields) that can take non-unique values generates an SQLCODE -314 error, with additional details available through %msg.

NO ACTION is the only referential action supported for sharded tables.

An ADD FOREIGN KEY is constrained when data already exists in the table. To change this default constraint behavior, refer to the COMPILEMODE=NOCHECK option of the SET OPTION command.

When you define an ADD FOREIGN KEY constraint for a single field and the foreign key references the idkey of the referenced table, InterSystems IRIS converts the property in the foreign key into a reference property. This conversion is subject to the following restrictions:

- The table must contain no data.

- The property on the foreign key cannot be of a persistent class (that is, it cannot already be a reference property).

- The data types and data type parameters of the foreign key field and the referenced idkey field must be the same.

- The foreign key field cannot be an IDENTITY field.

DROP CONSTRAINT Restrictions

By default, you cannot drop a unique or primary key constraint if it is referenced by a foreign key constraint. Attempting to do so results in an SQLCODE -317 error. To change this default foreign key constraint behavior, refer to the COMPILEMODE=NOCHECK option of the SET OPTION command.

The effects of dropping a primary key constraint depend on the setting of the Are Primary Keys also ID Keys setting (as described above):

- If the PrimaryKey index is not also the IDKey index, dropping the primary key constraint drops the index definition.

- If the PrimaryKey index is also the IDKey index, and there is no data in the table, dropping the primary key constraint drops the entire index definition.

- If the PrimaryKey index is also the IDKey index, and there *is* data in the table, dropping the primary key constraint just drops the PRIMARYKEY qualifier from the IDKey index definition.

DROP CONSTRAINT When Non-Existent

By default, InterSystems IRIS rejects an attempt to drop a field constraint on a field that does not have that constraint and issues an SQLCODE -315 error. To determine the current setting, call **$SYSTEM.SQL.CurrentSettings()**, which displays a `Allow DDL DROP of non-existent constraint` setting. The default is 0 (No), which is the recommended setting. If this option is set to 1 (Yes), an **ALTER TABLE** DROP CONSTRAINT causes InterSystems IRIS to perform no operation and issue no error message.

From the Management Portal, **System Administration**, **Configuration**, **SQL and Object Settings**, **SQL** you can set this option (and other similar create, alter, and drop options) system-wide by selecting the **Ignore redundant DDL statements** check box.

Examples

The following examples uses Embedded SQL programs to create a table, populate two rows, and then alter the table definition.

To demonstrate this, please run the first two Embedded SQL programs in the order shown. (It is necessary to use two embedded SQL programs here because embedded SQL cannot compile an **INSERT** statement unless the referenced table already exists.)

ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql(DROP TABLE SQLUser.MyStudents)
    IF SQLCODE=0 { WRITE !,"Deleted table" }
    ELSE { WRITE "DROP TABLE error SQLCODE=",SQLCODE }
&sql(CREATE TABLE SQLUser.MyStudents (
    FirstName VARCHAR(35) NOT NULL,
    LastName VARCHAR(35) NOT NULL)
    )
    IF SQLCODE=0 { WRITE !,"Created table" }
    ELSE { WRITE "CREATE TABLE error SQLCODE=",SQLCODE }
```

ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
NEW SQLCODE,%msg
&sql(INSERT INTO SQLUser.MyStudents (FirstName, LastName)
  VALUES ('David','Vanderbilt'))
IF SQLCODE=0 { WRITE !,"Inserted data in table"}
    ELSE { WRITE !,"SQLCODE=",SQLCODE,": ",%msg }
&sql(INSERT INTO SQLUser.MyStudents (FirstName, LastName)
  VALUES ('Mary','Smith'))
IF SQLCODE=0 { WRITE !,"Inserted data in table"}
    ELSE { WRITE !,"SQLCODE=",SQLCODE,": ",%msg }
```

The following example uses **ALTER TABLE** to add the ColorPreference column. Because the column definition specifies a default, the system populates ColorPreference with the value 'Blue' for the two existing rows of the table:

ObjectScript

```
  NEW SQLCODE,%msg
&sql(ALTER TABLE SQLUser.MyStudents
  ADD COLUMN ColorPreference VARCHAR(16) NOT NULL DEFAULT 'Blue')
IF SQLCODE=0 {
  WRITE !,"Added a column",! }
ELSEIF SQLCODE=-306 {
  WRITE !,"SQLCODE=",SQLCODE,": ",%msg }
ELSE { WRITE "SQLCODE error=",SQLCODE }
```

The following example uses **ALTER TABLE** to add two computed columns: FLName and LFName. For existing rows these columns have no value. For any subsequently inserted row a value is computed for each of these columns:

ObjectScript

```
NEW SQLCODE,%msg
&sql(ALTER TABLE SQLUser.MyStudents
  ADD COLUMN FLName VARCHAR(71) COMPUTECODE { SET {FLName}={FirstName}_" "_{LastName}}
            COMPUTEONCHANGE (FirstName,LastName),
      COLUMN LFName VARCHAR(71) COMPUTECODE { SET {LFName}={LastName}_ "," _{FirstName}}
            COMPUTEONCHANGE (FirstName,LastName) )
IF SQLCODE=0 {
  WRITE !,"Added two computed columns",! }
ELSE { WRITE "SQLCODE error=",SQLCODE }
```

See Also

- CREATE TABLE, DROP TABLE

- JOIN

- SELECT

- INSERT, UPDATE, INSERT OR UPDATE, DELETE

- Defining Tables

- SQL and Object Settings Pages

- SQLCODE error messages

ALTER USER (SQL)

Changes a user's password.

Synopsis

```
ALTER USER user-name IDENTIFY BY password
ALTER USER user-name IDENTIFIED BY password
```

Description

The **ALTER USER** command allows you to change a user's password. You can always change your own password. To change another user's password, you must be logged in as a user with one of the following:

- The %Admin_Secure administrative resource with USE permission.

- The %Admin_UserEdit administrative resource with USE permission.

- Full security privileges on the system.

The IDENTIFY BY and IDENTIFIED BY keywords are synonyms.

The *user-name* must be an existing user. Specifying a non-existent user generates an SQLCODE -400 error with a %msg such as the following: ERROR #838: User badname does not exist. You can determine if a user exists by invoking the **$SYSTEM.SQL.Security.UserExists()** method.

A *user-name* specified as a delimited identifier can be an SQL reserved word and can contain a comma (,), period (.), caret (^), and the two-character arrow sequence (->). It may begin with any valid character except the asterisk (*).

A *password* can be a string literal, a numeric, or an identifier. A string literal must be enclosed in quotes, and can contain any combination of characters, including blank spaces. A numeric or an identifier does not have to be enclosed in quotes. A numeric must consist of only the characters 0 through 9. An identifier must start with a letter (uppercase or lowercase) or a % (percent symbol); this can be followed by any combination of letters, numbers, or any of the following symbols: _ (underscore), & (ampersand), $ (dollar sign), or @ (at sign).

ALTER USER does not issue an error code if the new password is identical to the existing password. It sets SQLCODE = 0 (Successful Completion).

You can also change a user password using the **$SYSTEM.Security.ChangePassword()** method:

```
$SYSTEM.Security.ChangePassword(args)
```

Privileges

The **ALTER USER** command is a privileged operation. Prior to using **ALTER USER** in embedded SQL, you must be logged in as a user with either the %Admin_Secure administrative resource with USE permission, or the %Admin_UserEdit administrative resource with USE permission, or full security privileges on the system. Failing to do so results in an SQL-CODE -99 error (Privilege Violation). Use the **$SYSTEM.Security.Login()** method to assign a user with appropriate privileges:

ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql(       )
```

You must have the **%Service_Login:Use** privilege to invoke the **$SYSTEM.Security.Login** method. For further information, see %SYSTEM.Security.

Arguments

user-name

The name of an existing user whose password is to be changed. User names are not case-sensitive.

password

The new password for the user. A password must be at least 3 characters and cannot exceed 32 characters. Passwords are case-sensitive. Passwords can contain Unicode characters.

Examples

The following embedded SQL example changes the password of user Bill from "temp_pw" to "pw4AUser":

ObjectScript

```
Main
   DO $SYSTEM.Security.Login("_SYSTEM","SYS")
   &sql(CREATE USER Bill IDENTIFY BY temp_pw)
      IF SQLCODE=0 { WRITE !,"Created user" }
      ELSE { WRITE "CREATE USER error SQLCODE=",SQLCODE,! }
   &sql(ALTER USER BILL IDENTIFY BY pw4AUser)
      IF SQLCODE=0 { WRITE !,"Altered user password" }
      ELSE { WRITE "ALTER USER error SQLCODE=",SQLCODE,! }
Cleanup
   SET toggle=$RANDOM(2)
   IF toggle=0 {
     &sql(DROP USER Bill)
      IF SQLCODE=0 { WRITE !,"Dropped user" }
      ELSE { WRITE "DROP USER error SQLCODE=",SQLCODE }
   }
   ELSE {
     WRITE !,"No drop this time"
     QUIT
   }
```

See Also

- SQL statements: CREATE USER, DROP USER, GRANT, REVOKE

- SQL Users, Roles, and Privileges

- ObjectScript: $ROLES and $USERNAME special variables

- SQLCODE error messages

ALTER VIEW (SQL)

Modifies a view.

Synopsis

```
ALTER VIEW viewName AS query
ALTER VIEW viewName (column, column2, ...) AS query

ALTER VIEW viewName ... AS query WITH READ ONLY
ALTER VIEW viewName ... AS query WITH [LOCAL | CASCADED] CHECK OPTION
```

Description

The **ALTER VIEW** command modifies views created using the CREATE VIEW command or a view projected from a persistent class. The altered view replaces the existing view, so you cannot modify specific columns in a view.

A *view* is a virtual table based on the result set of a **SELECT** query or a **UNION** of such queries. For more details on views, see Defining and Using Views.

- **ALTER VIEW** *viewName* **AS** *query* replaces the existing columns in a view with the columns returned by the **SELECT** query. The view column names are derived from the column names returned by the result set of the query, which can be:

    - The column names or aliases of the table or view being queried

    - The column name of a class query defined as a table-valued function

    This statement modifies the `NewEmployees` view so that it includes only employees hired within the last 12 months. The view column names, `Name`, `Office`, and `StartDate`, match the column names of the source table.

    SQL

    ```
    ALTER VIEW NewEmployees AS
        SELECT Name,Office,StartDate
        FROM Sample.Employees
        WHERE DATEDIFF('month',StartDate,CURRENT_DATE) <= 12
    ```

- **ALTER VIEW** *viewName* **(***column***,** *column2***, ...) AS** *query* specifies the names of the columns to include in the view. The column names must correspond in number and sequence with the table columns returned by the **SELECT** query. Alternatively, you can specify these view column names as column name aliases in the **SELECT** statement query. These ALTER VIEW statements are equivalent:

    SQL

    ```
    ALTER VIEW MyView (MyViewCol1,MyViewCol2,MyViewCol3) AS
        SELECT TableCol1, TableCol2, TableCol3 FROM MyTable
    ```

    SQL

    ```
    ALTER VIEW MyView AS SELECT TableCol1 AS ViewCol1,
        TableCol2 AS ViewCol2,
        TableCol3 AS ViewCol3
        FROM MyTable
    ```

    The column specification replaces any existing columns specified for the view.

    Example: Create and Alter a View

- **ALTER VIEW** *viewName* **... AS** *query* **WITH READ ONLY** specifies that no insert, update, or delete operations can be performed through this view upon the table on which the view is based. The default is to permit these operations through a view, subject to any specified WITH CHECK OPTION constraints.

Example: Set Read-Only View

- **ALTER VIEW** *viewName* **... AS** *query* **WITH [LOCAL | CASCADED] CHECK OPTION** checks that any row being updated or inserted into this view satisfies the WHERE constraints of the view. If the row does not meet these constrains, that row is not updated or inserted. You can specify these check options:

    – WITH LOCAL CHECK OPTION — Check only the **WHERE** clause of the view specified in the **INSERT** or **UPDATE** statement.

    – WITH CASCADED CHECK OPTION or WITH CHECK OPTION — Check the **WHERE** clause of the view specified in the **INSERT** or **UPDATE** statement and all underlying views on which that view is based. This option overrides any WITH LOCAL CHECK OPTION clauses in these underlying views and is recommended for all updateable views.

    For more details on these options, see The WITH CHECK Option.

    Example: Validate Table Modifications Made Through a View

Arguments

viewName

The view being modified, which has the same naming rules as a table name. A view name can be qualified (schema.view-name), or unqualified (viewname). An unqualified view name takes the default schema name.

To determine if a specified view exists in the current namespace, use the **$SYSTEM.SQL.Schema.ViewExists()** method.

If the view is projected from a persistent class, you can run **ALTER VIEW** only if the view has the `Classtype="view"` and `DDLAllowed` keywords specified. You cannot alter views that are projected from a class query.

query

The result set from a query that serves as a the basis for the view. You can specify the query as a **SELECT** statement or a **UNION** of two or more **SELECT** statements. For an example that uses a UNION command, see Alter View Using Combined SELECT Queries.

A view query cannot contain host variables or include the INTO keyword. If you attempt to reference a host variable in *query*, the system generates an SQLCODE -148 error.

column

The name of a column included in the modified view. Specify multiple column names in a comma-separated list. You can specify column names after the *viewName* argument or in the *query* argument.

Examples

Create and Alter a View

This example shows how to create a view and then alter it. The example also shows how to query and delete the view.

Create a view the contains the names of people who live in Massachusetts. This example assumes that a `Sample.Person` table already exists and contains a `Home_State` column.

SQL

```
CREATE VIEW MassFolks (vFullName) AS
    SELECT Name FROM Sample.Person WHERE Home_State='MA'
```

You can then query the view as you would a regular table.

SQL

```
SELECT * FROM MassFolks
```

Modify the view to include new columns. Altering a view replaces the column list with a new column list but does not preserve the prior column list. Therefore, this modified view contains only the vMassAbbrev and vCity columns, not the vFullName column.

SQL

```
ALTER VIEW MassFolks (vMassAbbrev,vCity) AS
     SELECT Home_State,Home_City FROM Sample.Person WHERE Home_State='MA'
```

Delete the view. You can delete a view similar to how you would delete a regular table.

SQL

```
DROP VIEW MassFolks
```

Alter View Using Combined SELECT Queries

Alter a view to include the combined results of two SELECT queries. To combine the results, you use a UNION command.

SQL

```
ALTER VIEW MyView (vname,vstate) AS
  SELECT t1.name,t1.home_state
    FROM Sample.Person AS t1
  UNION
  SELECT t2.name,t2.office_state
    FROM Sample.Employee AS t2
```

Set Read-Only View

Modify a view to prevent modifying the underlying table through this view.

SQL

```
ALTER VIEW YoungPeople AS
     SELECT Name,DOB
     FROM Sample.Person
     WHERE DATEDIFF(year,DOB,CURRENT_DATE) <= 18
WITH READ ONLY
```

If you update any row through this view, the WITH READ ONLY prevents the update.

SQL

```
UPDATE YoungPeople (DOB)
VALUES (02/17/2022)
WHERE Name='Page,Laura O.'
```

Validate Table Modifications Made Through a View

Modify this view of honor students to prevent the insertion of students that do not meet the GPA criteria. This examples assumes that a Sample.Student table already exists.

SQL

```
ALTER VIEW HonorsStudent AS
    SELECT Name, GPA
    FROM Sample.Student
    WHERE GPA > 3.0
WITH CHECK OPTION
```

If you try to insert a student with too low a GPA for this view, the VIEW CHECK OPTION prevents the insertion.

SQL

```
INSERT INTO HonorsStudent (Name, GPA)
VALUES ('Waal,Edgar P.',2.9)
```

Security and Privileges

The **ALTER VIEW** command is a privileged operation. The user must have %ALTER_VIEW administrative privilege to execute **ALTER VIEW**. Failing to do so results in an SQLCODE –99 error with the %msg User 'name' does not have %ALTER_VIEW privileges.

The user must have %ALTER privilege on the specified view. If the user is the Owner (creator) of the view, the user is automatically granted %ALTER privilege for that view. Otherwise, the user must be granted %ALTER privilege for the view. Failing to do so results in an SQLCODE –99 error with the %msg User 'name' does not have privilege to ALTER the view 'Schema.ViewName'.

If you hold appropriate granting privileges, you can assign %ALTER_VIEW and %ALTER privileges by using the GRANT command.

To determine if the current user has %ALTER privileges, call the %CHECKPRIV command. To determine if a specified user has %ALTER privilege, call the **$SYSTEM.SQL.Security.CheckPrivilege()** method.

In embedded SQL, you can use the **$SYSTEM.Security.Login()** method to log in as a user with appropriate privileges:

```
DO $SYSTEM.Security.Login("myUserName","myPassword")
&sql(...)
```

You must have the **%Service_Login:Use** privilege to invoke the **$SYSTEM.Security.Login** method. For further information, see %SYSTEM.Security.

ALTER VIEW cannot be used on a view based on a table projected from a deployed persistent class. This operation fails with an SQLCODE -400 error with the %msg Unable to execute DDL that modifies a deployed class: 'classname'.

See Also

* CREATE VIEW, DROP VIEW, GRANT
* Defining Views
* SQLCODE error messages

BUILD INDEX (SQL)

Populates one or more indexes with data.

Synopsis

```
BUILD INDEX [%NOLOCK] [%NOJOURN] FOR TABLE table-name [INDEX index-name [,index-name]]

BUILD INDEX [%NOLOCK] [%NOJOURN] FOR SCHEMA schema-name

BUILD INDEX [%NOLOCK] [%NOJOURN] FOR ALL
```

Description

BUILD INDEX provides three syntax forms for building/re-building all defined indexes:

- Table: **BUILD INDEX FOR TABLE** *table-name*. The optional INDEX clause allows you to build/re-build only the specified indexes.

- All tables in a schema: **BUILD INDEX FOR SCHEMA** *schema-name*

- All tables in the current namespace: **BUILD INDEX FOR ALL**

You may wish to build indexes for any of the following reasons:

- You have used **CREATE INDEX** to add one or more indexes to a table that already contains data.

- You have performed **INSERT**, **UPDATE**, or **DELETE** operations on a table using the %NOINDEX option, rather than accepting the performance overhead of having each of these operations write to the index.

In either case, use **BUILD INDEX** to populate these indexes with data.

BUILD INDEX returns the number of tables modified as the number of Rows Affected.

If you used **CREATE INDEX** with the **DEFER BUILD** option to create an index, you must manually build the index. Note that the **BUILD INDEX** command builds the index's data, but does not make the index selectable, or usable, in queries. In order to make an index selectable, use the **SetMapSelectability()** method. You can view whether a map is selectable or not in the Management Portal by navigating to **System Explorer** > **SQL** > **Catalog Details** and selecting the **Maps/Indices** button.

Classes that were defined through ObjectScript may inherit indexes that need to be built from a superclass. To build these "inherited" indexes, you must call **BUILD INDEX** on the superclass that defines the index, not on the subclass that uses it.

If a table uses %Storage.SQL, then indexes explicitly defined within the class will not be built.

Privileges

The **BUILD INDEX** command is a privileged operation. The user must have %BUILD_INDEX administrative privilege to execute **BUILD INDEX**. Failing to do so results in an SQLCODE –99 error with the %msg `User 'name' does not have %BUILD_INDEX privileges`. You can use the GRANT command to assign %BUILD_INDEX privileges to a user or role, if you hold appropriate granting privileges. Administrative privileges are namespace-specific. For further details, refer to Privileges.

The user must have SELECT privilege on the specified table. If the user is the Owner (creator) of the table, the user is automatically granted SELECT privilege for that table. Otherwise, the user must be granted SELECT privilege for the table.

- Issuing **BUILD INDEX FOR TABLE** without SELECT privilege on the specified table results in an SQLCODE –30 error with the %msg `Table 'name' not found`.

- Issuing **BUILD INDEX FOR SCHEMA** only builds indexes for those table for which the user has SELECT privilege. If the user does not have SELECT privilege for any tables in the schema, the command completes without error, with 0 rows affected.

You can determine if the current user has SELECT privilege by invoking the %CHECKPRIV command. You can use the GRANT command to assign SELECT privilege to a specified table. For further details, refer to Privileges.

Locking and Journaling

By default, the **BUILD INDEX** statement acquires an extent lock on each table prior to building its indexes. This prevents other processes from modifying the table's data. This lock is automatically released at the conclusion of the **BUILD INDEX** operation. You can specify %NOLOCK to prevent table locking.

By default, the **BUILD INDEX** statement uses the journaling setting for the current process. You can specify %NOJOURN to prevent journaling.

To use %NOLOCK or %NOJOURN, you must have the corresponding SQL administrative privilege, which you can set by using the GRANT command.

Error Codes

- If the specified *table-name* does not exist, InterSystems IRIS issues an SQLCODE -30 error and sets %msg to `Table 'sample.tname' does not exist`. This error message is returned if you specify a view rather than a table, or if you specify a table for which you do not have SELECT privilege.

- If the specified *index-name* does not exist, InterSystems IRIS issues an SQLCODE -400 error and sets %msg to `ERROR #5066: Index name 'sample.tname::badindex' is invalid`.

- If the specified *schema-name* does not exist, InterSystems IRIS issues an SQLCODE -473 error and sets %msg to `Schema 'sample' not found`.

Arguments

FOR TABLE table-name

The name of an existing table. A *table-name* can be qualified (schema.table), or unqualified (table). An unqualified table name takes the default schema name.

INDEX index-name

An optional index name or a comma-separated list of index names. If specified, only these indexes are built. If not specified, all indexes defined for the table are built.

FOR SCHEMA schema-name

The name of an existing schema. This command builds all indexes for all tables in the specified schema.

See Also

- CREATE INDEX
- Defining and Building Indices
- SQLCODE error messages

CALL (SQL)

Invokes a stored procedure.

Synopsis

```
CALL procname(arg_list) [USING contextvar]

retval=CALL procname(arg_list) [USING contextvar]
```

Description

A **CALL** statement invokes a query exposed as an SQL stored procedure. The *procname* must be an existing stored procedure in the current namespace. If InterSystems IRIS cannot locate *procname*, it generates an SQLCODE -428 error. The *procname* must be a Stored Procedure with SqlProc=True. Refer to SqlProc.

For further details on stored procedures, refer to the CREATE PROCEDURE command.

Arguments

procname

The name of an existing stored procedure. The *procname* must be followed by parentheses, even if no arguments are specified. A procedure name can take any of of the following forms:

- Unqualified: Takes the default schema name. For example, `MedianAgeProc()`.

- Qualified: Supplies a schema name. For example, `Patient.MedianAgeProc()`.

- Multilevel: Qualified with one or more schema levels to paralell corresponding class package members. In this case, the *procname* may contain only one period character; the other periods in the corresponding class method name are replaced with underline characters. The period is specified before the lowest level class package member. For example, `%SYSTEM.SQL_GetROWID()`, or `%SYS_PTools.StatsSQL_Export()`.

InterSystems IRIS locates the match for an unqualified *procname* in a schema, using either the default schema name, or (if provided) a schema name from the schema search path. If InterSystems IRIS cannot locate the specified procedure using either the schema search path or the system-wide schema default, it generates an SQLCODE -428 error. You can use the **$SYSTEM.SQL.Schema.Default()** method to determine the current system-wide default schema name. The initial system-wide default schema name is `SQLUser`, which corresponds to the class package name `User`.

To determine if a *procname* exists in the current namespace, use the **$SYSTEM.SQL.Schema.ProcedureExists()** method. The *procname* is not case-sensitive.

You must append the argument parentheses to the *procname*, even if you are not specifying any arguments. Failing to do so results in an SQLCODE -1 error.

arg_list

A list of arguments used to pass values to the stored procedure. The *arg_list* is enclosed in parentheses and arguments in the list are separated by commas. The parentheses are mandatory, even if you specify no arguments.

The *arg_list* arguments are optional. This comma-separated list is known as the actual argument list, which must match in number and in sequence the formal argument list in the procedure definition. You may specify fewer actual argument values than the formal arguments defined in the stored procedure. If you specify more actual argument values than the formal arguments defined in the stored procedure, the system generates an SQLCODE -370 error. This error message specifies the name of the stored procedure, the number of arguments specified, and the number of arguments defined in the stored procedure.

You can omit trailing arguments; any missing trailing arguments are undefined and take default values. You can specify an undefined argument within the argument list by specifying a placeholder comma. For example, (*arg1*,,*arg3*) passes three arguments, the second of which is undefined. Commonly, undefined arguments take a default value that was specified when defining the stored procedure. If no default is defined, an undefined argument takes NULL. For further details refer to NULL and the Empty String.

If you specify an argument value that does not match the data type defined in the stored procedure that argument takes NULL, even if a default value is defined. For example, a stored procedure defines an argument as `IN numarg INT DEFAULT 99`. If **CALL** specifies a numeric argument, that *arg* value is used. If **CALL** omits the argument, the defined default is used. However, if **CALL** specifies a non-numeric argument, NULL is used, not the defined default.

An *arg_list* argument can be a user-defined function (a method stored procedure that returns a value).

USING contextvar

An optional argument. *contextvar* specifies a descriptor area variable that receives the procedure context object generated by the procedure call. If omitted, the default is %sqlcontext.

retval

An optional variable specified to receive the procedure return value. Can contain a single value, not a result set. Can be specified as a local variable, a host variable, or a question mark (?) argument.

From Embedded SQL

ObjectScript embedded SQL can either issue a **CALL** statement, or use the **DO** command to invoke the underlying routine or method.

Using Embedded SQL, you can supply argument values to **CALL** as literals or by using any combination of `:name` host variables or question mark (?) input parameters, as follows:

ObjectScript

```
SET a=7,b="A",c=99
&sql(CALL MyProc(:a,:b,:c))
```

ObjectScript

```
&sql(CALL MyProc(?,:b,?))
```

The initial invocation of a **CALL** statement in Embedded SQL creates an %sqlcontext variable, by default. Subsequent iterations use this existing %sqlcontext variable, meaning multiple iterations accumulate results in %sqlcontext that could potentially result in a <STORE> error. If a **CALL** statement is to be iterated repeatedly, you can explicitly specify the %sqlcontext variable in the USING clause. When a procedure context is specified in the USING clause InterSystems IRIS issues a NEW on that procedure context each time it is invoked.

A host variable used for an output *arg* can be a single value, an array reference, an oref.property reference, or a multidimensional oref.property reference.

You can return a value from a **CALL** statement by using either a host variable or a question mark (?):

ObjectScript

```
&sql(:rtnval=CALL MyProc())
```

ObjectScript

```
&sql(?=CALL MyProc())
```

The **CALL** return value must be a single value. You cannot return a result set from a **CALL** statement in Embedded SQL. Attempting to use `retval=CALL` syntax for a procedure that does not return a value generates an SQLCODE -371 error.

For further details, refer to Embedded SQL.

From Dynamic SQL

The following Dynamic SQL example calls the Stored Procedure Sample.PersonSets, which performs two queries on the Sample.Person table. The Stored Procedure arguments specify the WHERE clause values for these two queries. The first argument specifies to return all records in the first query where Name starts with *arg1* (in this case, the letter "M"). The second argument specifies to return all records in the second query where Home_State = *arg2* (in this case, "MA"):

ObjectScript

```
SET mycall = "CALL Sample.PersonSets(?,'MA')"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(mycall)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute("M")
IF rset.%SQLCODE '= 0 {WRITE "SQL error=",rset.%SQLCODE  QUIT}
DO rset.%Display()
```

The following Dynamic SQL example also calls the Stored Procedure Sample.PersonSets, returning the result sets for each query separately. The **%Next()** method iterates through the first query result set. The **%MoreResults()** method accesses the result set for the second query. If there were more than two queries, **%MoreResults()** would access each result set in turn.

ObjectScript

```
#include %occStatus
set mycall = "CALL Sample.PersonSets(?,'MA')"
set tStatement = ##class(%SQL.Statement).%New()
set qStatus = tStatement.%Prepare(mycall)
if $$$ISERR(qStatus) {write "%Prepare failed:" do $SYSTEM.Status.DisplayError(qStatus) quit}

set rset = tStatement.%Execute("M")
if (rset.%SQLCODE '= 0) {write "%Execute failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message
 quit}

FirstResultSet
  while rset.%Next()
  {
    write "Name: ",rset.%Get("Name")
    if rset.%Get("Spouse") {write " Spouse: ",rset.%Get("Spouse"),!}
    else {write " unmarried",!}
  }
  if (rset.%SQLCODE < 0) {write "%Next failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message
quit}

  write !,"1st row count=",rset.%ROWCOUNT,!!

SecondResultSet
  while rset.%MoreResults()
  {
    do rset.%CurrentResult.%Display()
  }
```

Note that it is important to check the %SQLCODE value set by the **CALL** execution before invoking **%Next()**. Invoking the **%Next()** method sets %SQLCODE, overwriting the prior **CALL** %SQLCODE value. If **%Next()** receives no result set data, it sets %SQLCODE=100. It does not distinguish between an empty result set (no rows selected) and a nonexistent result set due to an error in **CALL** processing.

For further details on %SQL.Statement and on how to display a list of formal parameters and other metadata for a stored procedure, refer to Using Dynamic SQL. Also, Returning the Full Result Set provides further information and examples of the **%Display()** method. Returning Specific Values from the Result Set provides further information and examples of the **%Next()** and **%Get()** methods.

From ObjectScript

Rather than calling stored procedures directly from embedded SQL, you can invoke stored procedures through ObjectScript calls to the class methods that contain them. In this case, you have to manage the parameters, and with query-based stored procedures, the separate methods have to be called and the fetch loop managed.

For example, to call a method exposed as a stored procedure called UpdateAllAvgScores that has no arguments, the code is:

ObjectScript

```
NEW phnd
SET phnd=##class(%SQLProcContext).%New()
DO ##class(students).UpdateAllAvgScores(phnd)
IF phnd.%SQLCODE {QUIT phnd.%SQLCODE}
USE 0
WRITE !,phnd.%ROWCOUNT," Rows Affected"
```

When specifying a procedure's arguments in the call statement, you must *not* specify the %Library.SQLProcContext parameter if the procedure has an explicitly defined %Library.SQLProcContext parameter. The handling of this parameter is done automatically.

In the following example, the stored procedure takes two arguments. It has an explicitly defined procedure context.

ObjectScript

```
NEW phnd
SET phnd=##class(%SQLProcContext).%New()
SET rtn=##class(Sample.ResultSets).PersonSets("D","NY")
IF phnd.%SQLCODE {QUIT phnd.%SQLCODE}
DO %sqlcontext.%Display()
WRITE !,"All Done"
```

To call a stored procedure that has been implemented as a query, you must call all three methods:

ObjectScript

```
NEW qhnd
DO ##class(students).GetAvgScoreExecute(.qhnd,x1)
NEW avgrow,AtEnd
SET avgrow=$lb("")
SET AtEnd=0
DO ##class(students).GetAvgScoreFetch(.qhnd,.avgrow,.AtEnd)
SET x5=$lg(avgrow,1)
DO ##class(students).GetAvgScoreClose(qhnd)
```

If a query-based stored procedure is to be nested within a number of other stored procedures, it is useful to write a wrapper method to hide all of this.

From ODBC or JDBC

InterSystems IRIS fully supports **CALL** syntax as defined by the ODBC 2.x and JDBC 1.0 standards. In JDBC, you can invoke **CALL** through the methods of the CallableStatement class. In ODBC, there are APIs. The **CALL** syntax and semantics are exactly the same for JDBC and ODBC. Further, they are processed in the same way: both drivers parse the statement text and, if the statement is **CALL**, they directly invoke the special methods on the server side, bypassing the SQL engine.

If class PERSON has a stored procedure called SP1, you can call this from an ODBC or JDBC client (such as Microsoft Query) as follows:

```
retcode = SQLExecDirect(hstmt, "{?=call PERSON_SP1(?,?)}", SQL_NTS);
```

InterSystems IRIS conforms to the ODBC standard in its structure for calling stored procedures. See the relevant documentation for more information on that standard.

With ODBC only, InterSystems IRIS allows relaxed syntax for calls, so there does not need to be curly braces around **CALL** or parentheses around parameters. (Since this is good programming form, the above example uses them.)

Again, with ODBC only, InterSystems IRIS allows modified syntax for using default parameters, so that **CALL SP** is different from **CALL SP()**. The second form implies passing of a default parameter — as does `CALL SP (,,)` or `SP(,?,)` or other such syntax. In that sense, the parenthesized form of **CALL** is different from non-parenthesized.

See Also

- SQL statements: CREATE PROCEDURE, CREATE QUERY, CREATE METHOD

- ObjectScript: DO command

- Defining and Using Stored Procedures

- SQLCODE error messages

CASE (SQL)

Chooses one of a specified set of values depending on some condition.

Synopsis

```
CASE    WHEN search_condition THEN value_expression  [ WHEN search_condition THEN value_expression
... ]  [ ELSE value_expression ] END  CASE value_expression   WHEN value_expression THEN value_expression
  [ WHEN value_expression THEN value_expression ... ]   [ ELSE value_expression ] END
```

Arguments

Argument	Description
search_condition	An SQL boolean expression.
value_expression	An SQL expression (such as a literal value or field name.)

Description

The **CASE** expression allows you to make comparison tests on series of values, returning when it encounters the first match.

The **CASE** expression comes in two forms: Simple and Searched.

The Simple **CASE** expression tests a series of value *expressions* (specified by a **WHEN** clause) to see if they are equal to a given value expression:

SQL

```
SELECT
CASE Field1
   WHEN 1 THEN 'ONE'
   WHEN 2 THEN 'TWO'
   ELSE NULL
END
FROM MyTable
```

The value associated with the first matching expression is returned as the value of the **CASE** expression.

Numeric *value_expression* values may have different data types. The data type returned is the type most compatible with all of the possible result values, the data type with the highest data type precedence. For numeric *value_expression* values **CASE** returns the largest length, precision, and scale from all of the possible result values. A result value of NULL has the lowest data type precedence; however, if all result values are NULL, the data type returned is VARCHAR.

The Searched **CASE** expression tests a series of search *conditions* (specified by a **WHEN** clause), finds the first **WHEN** condition that evaluates to true, and returns the value associated with it:

SQL

```
SELECT
CASE
   WHEN Field1 = 1 THEN 'ONE'
   WHEN Field1 = 2 THEN 'TWO'
   ELSE NULL
END
FROM MyTable
```

With either form of **CASE** expression, you can use an **ELSE** clause to specify what value to return if none of the **WHEN** clause conditions are true. If you omit the **ELSE** clause and none of the **WHEN** clause conditions are true, **CASE** returns NULL.

A **CASE** comparison that tests for NULL must use the IS NULL or IS NOT NULL keyword phrase. NULL is *not* a data value (it represents the absence of a value). For this reason, any equality or arithmetic test for NULL always returns false. A **CASE** expression that compares NULL and any data value always returns false. For example, NULL < 1 and NULL > 1 both return false. A **CASE** expression that equates NULL with NULL also returns false.

The end of a **CASE** expression is marked by an END token.

Examples

The following query is an example of a Simple **CASE** expression, where specified field values are replaced by supplied values. Note the use of the RetireAge column alias after the END keyword; the optional AS keyword is omitted in this example:

SQL

```
SELECT Name,
CASE Age
   WHEN 65 THEN 'Retire this year'
   WHEN 64 THEN 'Retire next year'
   ELSE 'Past retirement age '|| Age
END RetireAge
FROM Sample.Person
WHERE Age > 63
ORDER BY Age
```

The following query is another example of a Simple **CASE** expression. This query labels rows with certain Home_State values as either "Northern NE" or "Southern NE", and sets all other Home_State values in this column to NULL. It uses the As clause to label this column as "NewEnglanders", and also displays Names and the original Home_State values. The resulting rows are ordered first by the NewEnglanders column (in descending order), and within this alphabetically by Home_State, and then by Name.

SQL

```
SELECT Name,
CASE Home_State
   WHEN 'VT' THEN 'Northern NE'
   WHEN 'NH' THEN 'Northern NE'
   WHEN 'ME' THEN 'Northern NE'
   WHEN 'MA' THEN 'Southern NE'
   WHEN 'CT' THEN 'Southern NE'
   WHEN 'RI' THEN 'Southern NE'
   ELSE NULL
END AS NewEnglanders, Home_State
FROM Sample.Person
ORDER BY NewEnglanders DESC,Home_State,Name
```

The following query is an example of a Searched **CASE** expression. It uses logical operators (greater than (>), logical AND (&), logical OR (!)) to specify a boolean statement for each **WHEN** clause. The first **WHEN** clause that tests True sets the value expression that follows the **THEN** keyword. In this example, the Age and Home_State field values are used to identify three types of Yankees: Old Yankees, Yankees (residents of the six New England states), and likely fans of the New York Yankees baseball team:

SQL

```
SELECT Name,
CASE
WHEN Age > 55 & Home_State = 'VT'
   ! Home_State='ME' ! Home_State='NH'
   ! Home_State='MA' ! Home_State='CT'
   ! Home_State='RI'
THEN 'Old Yankee'
WHEN Home_State = 'VT'
   ! Home_State='ME' ! Home_State='NH'
   ! Home_State='MA' ! Home_State='CT'
   ! Home_State='RI'
THEN 'Yankee'
WHEN Home_State='NY' THEN 'Yankees Fan'
   ELSE Home_State
END AS Yankees
FROM Sample.Person
```

The following example shows that any comparison with NULL always returns false:

SQL

```
SELECT TOP 5 Name,
CASE NULL
   WHEN NULL THEN 'Null = Null'
   WHEN 0 THEN 'Null = 0'
   WHEN '' THEN 'Null = empty string'
   WHEN CHAR(0) THEN 'Null = CHAR(0)'
   ELSE 'Null Arithmetic Invalid'
END
FROM Sample.Person
```

The following example shows how to use **CASE** with a field that has NULLs:

SQL

```
SELECT TOP 20 Name,
CASE
   WHEN FavoriteColors IS NULL THEN 'No Colors'
   ELSE $LISTTOSTRING(FavoriteColors,':')
END
FROM Sample.Person
```

CASE is not limited to use in queries, as shown in the following example:

SQL

```
INSERT INTO SQLUser.MyStudents (Name, PxTs) VALUES (
CASE ?
    WHEN 'a' THEN 'Alice'
    WHEN 'b' THEN 'Barney'
    ELSE 'Unknown' END,
CURRENT_TIMESTAMP)
```

See Also

- SQL functions: DECODE, GREATEST, LEAST, NULLIF, COALESCE

- ObjectScript function: $CASE

%CHECKPRIV (SQL)

Checks whether the user holds a specified privilege.

Synopsis

```
%CHECKPRIV [GRANT OPTION FOR | ADMIN OPTION FOR] syspriv [,syspriv]
%CHECKPRIV [GRANT OPTION FOR] objpriv
      ON object
%CHECKPRIV column-privilege (column-list)
      ON table
```

Description

%CHECKPRIV can be used in two ways:

- To determine if the current user holds a specified system privilege, or holds all of the system privileges specified in a comma-separated list.

- To determine if the current user holds a user privilege of a specified type on a specified object. These objects can include table-level privileges on tables or views, column-level privileges on specified columns, and privileges on stored procedures.

If the user holds the specified privilege, **%CHECKPRIV** sets SQLCODE=0. If the user does not hold the specified privilege, **%CHECKPRIV** sets SQLCODE=100.

%CHECKPRIV enables you to check whether a privilege is held. It does not enforce privileges:

- Embedded SQL does not enforce privileges. **%CHECKPRIV** is primarily used for Embedded SQL. See Embedded SQL and Privileges.

- Dynamic SQL enforces privileges at runtime. For example, if you do not have the %CREATE_TABLE system privilege, **%CHECKPRIV %CREATE_TABLE** sets SQLCODE=100, showing that you don't have this privilege. Dynamic SQL enforces this privilege; a **CREATE TABLE** operation fails with an SQLCODE -99 error.

  At runtime, Dynamic SQL and ODBC/JDBC enforce privileges and generate appropriate errors. The Management Portal **Execute Query** SQL interface and the SQL Shell both execute as Dynamic SQL.

Because **%CHECKPRIV** requires access to the SQLCODE 100 value (an SQLCODE status value, not an SQLCODE error value) to determine its result, **%CHECKPRIV** cannot be directly used by JDBC and other clients that can only distinguish error or no error status.

Because **%CHECKPRIV** prepares and executes quickly, and is generally run only once, InterSystems IRIS does not create a cached query for **%CHECKPRIV**.

The CheckPrivilege() Method

The **$SYSTEM.SQL.Security.CheckPrivilege**() method provides greater functionality for checking user privileges on a table, view, or stored procedure:

- **CheckPrivilege()** checks privileges for a specified user. **%CHECKPRIV** only checks privileges for the current user.

- **CheckPrivilege()** allows you to check multiple privileges. Each invocation of **%CHECKPRIV** can only check one *objpriv* privilege.

- **CheckPrivilege()** allows you to check privileges on a table, view, or procedure defined in another namespace. **%CHECKPRIV** only checks privileges for objects in the current namespace.

Embedded SQL and Privileges

Privileges are not automatically checked or enforced for Embedded SQL. Therefore, an Embedded SQL program should (in most cases) call **%CHECKPRIV** before attempting a privileged operation, such as an update:

ObjectScript

```
SET name="Fred",age=25
SET SQLCODE=""
&sql(%CHECKPRIV UPDATE ON Sample.Person)
IF SQLCODE=100 {
    WRITE !,"No UPDATE privilege"
    QUIT }
ELSEIF SQLCODE < 0 {
    WRITE !,"Unexpected SQL error: ",SQLCODE," ",%msg
    QUIT }
ELSE {
    WRITE !,"Proceeding with UPDATE" }
&sql(UPDATE Sample.Person SET Name=:name,Age=:age WHERE Address='123 Bedrock')
IF SQLCODE=0 { WRITE !,"UPDATE successful" }
ELSE { WRITE "UPDATE error SQLCODE=",SQLCODE }
```

Arguments

GRANT OPTION FOR

This optional keyword phrase specifies checking whether the current user holds the WITH GRANT OPTION privilege on the specified privilege(s). A **%CHECKPRIV** with this option *does not* check whether the user holds the specified privilege(s) itself.

ADMIN OPTION FOR

This optional keyword phrase specifies checking whether the current user can grant the specified system privilege(s) to other users or roles. A **%CHECKPRIV** with this option *does not* check whether the user holds the specified privilege(s) itself.

syspriv

A system privilege, or a comma-separated list of system privileges. The available *syspriv* options include sixteen object definition privileges and four data modification privileges.

The object definition privileges are: %CREATE_FUNCTION, %DROP_FUNCTION, %CREATE_METHOD, %DROP_METHOD, %CREATE_PROCEDURE, %DROP_PROCEDURE, %CREATE_QUERY, %DROP_QUERY, %CREATE_TABLE, %ALTER_TABLE, %DROP_TABLE, %CREATE_VIEW, %ALTER_VIEW, %DROP_VIEW, %CREATE_TRIGGER, %DROP_TRIGGER. Alternatively, you can specify %DB_OBJECT_DEFINITION, which tests all 16 object definition privileges.

The data modification privileges are the %NOCHECK, %NOINDEX, %NOLOCK, %NOTRIGGER privileges for INSERT, UPDATE, and DELETE operations.

objpriv

An object privilege associated with a specified *object*. The available options are: %ALTER, DELETE, SELECT, INSERT, UPDATE, EXECUTE, and REFERENCES.

object

The name of the object for which the *objpriv* is being checked.

column-privilege

A column-level privilege associated with one or more listed columns. Available options are SELECT, INSERT, UPDATE, and REFERENCES.

column-list

A list of one or more column names for which privilege assignment is being checked, separated by commas and enclosed in parentheses. A space may be included or omitted between the *column-privilege* name and the opening parenthesis.

table

The name of the table or view that contains the *column-list* columns. A table name or view name can be qualified (schema.tablename), or unqualified (tablename). An unqualified name takes the default schema name; a schema search path is ignored.

Examples

The following Embedded SQL example checks whether the current user holds a specific object privilege for a specific table:

ObjectScript

```
&sql(%CHECKPRIV UPDATE ON Sample.Person)
IF SQLCODE=0 {WRITE "Have update privilege"}
ELSEIF SQLCODE=100 {WRITE "Do not have update privilege" QUIT}
ELSE {WRITE "Unexpected %CHECKPRIV error: ",SQLCODE," ",%msg  QUIT}
```

The following Embedded SQL example checks whether the current user holds system privileges on the three table operations. If it has privileges, it creates a table:

ObjectScript

```
&sql(%CHECKPRIV %CREATE_TABLE,%ALTER_TABLE,%DROP_TABLE)
IF SQLCODE=0 {WRITE "Have table privileges",!}
ELSEIF SQLCODE=100 {WRITE "Do not have one or more table privileges"  QUIT}
ELSE {WRITE "Unexpected %CHECKPRIV error: ",SQLCODE," ",%msg  QUIT}
&sql(CREATE TABLE Sample.MyTable (Name VARCHAR(40),Age INTEGER))
WRITE "Created table"
```

The following Embedded SQL example checks whether the current user holds all 16 object definition privileges. The SQLCODE value is set to either 0 (holds all 16 privileges) or 100 (does not hold one or more of the 16 privileges):

ObjectScript

```
&sql(%CHECKPRIV %DB_OBJECT_DEFINITION)
IF SQLCODE=0 {WRITE "Have all system privileges"}
ELSEIF SQLCODE=100 {WRITE "Do not have one or more system privileges"}
ELSE {WRITE "Unexpected SQLCODE error: ",SQLCODE," ",%msg}
```

The following Embedded SQL example checks whether the current user can grant the %CREATE_TABLE privilege to other users or roles:

ObjectScript

```
&sql(%CHECKPRIV ADMIN OPTION FOR %CREATE_TABLE)
IF SQLCODE=0 {WRITE "Have admin option on privilege"}
ELSEIF SQLCODE=100 {WRITE "Do not have admin option on privilege"}
ELSE {WRITE "Unexpected SQLCODE error: ",SQLCODE," ",%msg}
```

The following Embedded SQL example checks whether the current user holds the specified column-level privileges. Following the name of the privilege, specify the name of a column (or a comma-separated list of columns) in parentheses:

ObjectScript

```
&sql(%CHECKPRIV UPDATE(Name,Age) ON Sample.Person)
IF SQLCODE=0 {WRITE "Have privilege on all specified columns"}
ELSEIF SQLCODE=100 {WRITE "Do not have privilege on one or more specified columns"}
ELSE {WRITE "Unexpected SQLCODE error: ",SQLCODE," ",%msg}
```

See Also

- SQL statements: GRANT, REVOKE

- SQL Users, Roles, and Privileges

- ObjectScript: $ROLES and $USERNAME special variables

CLOSE (SQL)

Closes a cursor.

Synopsis

```
CLOSE cursor-name
```

Description

A **CLOSE** statement shuts down an open cursor. It releases the current result set and frees any cursor locks held on the rows on which the cursor is positioned. However, **CLOSE** does not delete the cursor; it leaves the data structures accessible for reopening, but fetches and positioned updates are not allowed until the cursor is reopened. This behavior is demonstrated by the following command sequences:

- **DECLARE c1**, **OPEN c1**, **FETCH c1**, **CLOSE c1** is the standard sequence.

- **DECLARE c1**, **OPEN c1**, **CLOSE c1**, **OPEN c1** reopens the declared cursor c1.

- **DECLARE c1**, **OPEN c1**, **CLOSE c1**, **DECLARE c1**, **OPEN c1** reopens the cursor specified in the first **DECLARE**, the second **DECLARE** is ignored.

- **DECLARE c1**, **OPEN c1**, **FETCH c1**, **CLOSE c1**, **OPEN c1**, **FETCH c1** cause both fetch operations to retrieve the same record.

CLOSE must be issued on an open cursor. Issuing a **CLOSE** on a cursor that has only been declared (but not opened), or on a cursor that has already been closed results in an SQLCODE -102 error. Issuing a **CLOSE** on a non-existent cursor — for example, a cursor that differs from the defined cursor in letter case — results in an SQLCODE -52 error.

The *cursor-name* is not namespace-specific. Changing the current namespace has no effect on use of a declared cursor. The only namespace consideration is that **FETCH** must occur in the namespace that contains the table(s) being queried.

Note that, as an SQL statement, **CLOSE** is only supported from Embedded SQL. Equivalent operations are supported through ODBC using the ODBC API.

Arguments

cursor-name

The name of the cursor to be closed. The cursor name was specified in the **DECLARE** statement. Cursor names are case-sensitive.

Examples

The following Embedded SQL example shows a cursor (named EmpCursor) being opened and closed:

ObjectScript

```
SET name="LastName,FirstName",state="##"
&sql(DECLARE EmpCursor CURSOR FOR
     SELECT Name, Home_State
     INTO :name,:state FROM Sample.Employee
     WHERE Home_State %STARTSWITH 'A')
WRITE !,"BEFORE: Name=",name," State=",state
&sql(OPEN EmpCursor)
  IF SQLCODE<0 {WRITE "SQL Open Cursor Error:",SQLCODE," ",%msg  QUIT}
NEW %ROWCOUNT,%ROWID
FOR { &sql(FETCH EmpCursor)
     QUIT:SQLCODE
     WRITE !,"DURING: Name=",name," State=",state }
WRITE !,"After FETCH SQLCODE: ",SQLCODE
WRITE !,"After FETCH row count: ",%ROWCOUNT
&sql(CLOSE EmpCursor)
   IF SQLCODE<0 {WRITE "SQL Close Cursor Error:",SQLCODE," ",%msg  QUIT}
WRITE !,"After CLOSE SQLCODE: ",SQLCODE
WRITE !,"After CLOSE row count: ",%ROWCOUNT
WRITE !,"AFTER: Name=",name," State=",state
```

Note that after closing the cursor, the host variables remain set to the last fetched data values, and %ROWCOUNT remains set to the number of rows retrieved. However, the SQLCODE value at the end of the fetch (SQLCODE=100) is overwritten by the SQLCODE value for the **CLOSE** (SQLCODE=0).

The following Embedded SQL example shows that a cursor persists across namespaces. This cursor is declared in %SYS, opened and fetched in USER, and closed in SAMPLES. Note that the **OPEN** must be executed in the namespace that contains the table(s) being queried, and the **FETCH** must able to access the output host variables, which are namespace-specific:

```
  &sql(USE DATABASE %SYS)
  WRITE $ZNSPACE,!
&sql(DECLARE NSCursor CURSOR FOR SELECT Name INTO :name FROM Sample.Employee)
  &sql(USE DATABASE "USER")
  WRITE $ZNSPACE,!
&sql(OPEN NSCursor)
     IF SQLCODE<0 {WRITE "SQL Open Cursor Error:",SQLCODE," ",%msg  QUIT}
   NEW SQLCODE,%ROWCOUNT,%ROWID
FOR { &sql(FETCH NSCursor)
     QUIT:SQLCODE
     WRITE "Name=",name,! }
  &sql(USE DATABASE SAMPLES)
  WRITE $ZNSPACE,!
&sql(CLOSE NSCursor)
   IF SQLCODE<0 {WRITE "SQL Close Cursor Error:",SQLCODE," ",%msg  QUIT}
```

See Also

- DECLARE, FETCH, OPEN

- SQL Cursors

COMMIT (SQL)

Commits work performed during a transaction.

Synopsis

```
COMMIT [WORK]
```

Description

A **COMMIT** statement commits all work completed during the current transaction, resets the transaction level counter, and releases all locks established. This completes the transaction. Work committed cannot be rolled back.

COMMIT and **COMMIT WORK** are equivalent statements; both versions are supported for compatibility.

A transaction is defined as the operations that have occurred since and including the **START TRANSACTION** statement. A **COMMIT** restores the transaction level counter (*$TLEVEL*) to its state immediately prior to the **START TRANSACTION** statement that initialized the transaction. (Because InterSystems SQL does not support nested transactions, issuing additional **START TRANSACTION** statements within a transaction has no effect on the transaction initialization point.)

A single **COMMIT** causes all savepoints within the transaction to be committed.

A **START TRANSACTION** statement is used to explicitly begin a new transaction. However, use of **START TRANSACTION** is optional. If transaction processing is activated, the first database operation following a **COMMIT** implicitly begins a new transaction. A **COMMIT** statement is not meaningful if either transaction processing is not in effect, or transaction processing is in effect with automatic commits. If no transaction is in progress, a **COMMIT** completes successfully (SQLCODE 0), but performs no operation.

The effects of a **COMMIT** on queries are determined by the current isolation level. These transaction parameters can be set using either the **SET TRANSACTION** or **START TRANSACTION** command.

An SQLCODE -400 is issued if a transaction operation fails to complete successfully.

ObjectScript and SQL Transactions

ObjectScript and SQL transaction commands are fully compatible and interchangeable, with the following exception:

ObjectScript **TSTART** and SQL **START TRANSACTION** both start a transaction if no transaction is current. However, **START TRANSACTION** does not support nested transactions. Therefore, if you need (or may need) nested transactions, it is preferable to start the transaction with **TSTART**. If you need compatibility with the SQL standard, use **START TRANSACTION**.

ObjectScript transaction processing provides limited support for nested transactions. SQL transaction processing supplies support for savepoints within transactions.

If a transaction involves SQL update statements, the transaction should be started by the SQL **START TRANSACTION** statement and committed with the SQL **COMMIT** statement. Methods that use **TSTART**/**TCOMMIT** nesting can be included in the transaction, as long as they don't initiate the transaction. Methods and stored procedures should not normally use SQL transaction control statements, unless, by design, they are the main controller of the transaction. Stored procedures should not normally use SQL transaction control statements, because these stored procedures are normally called from ODBC/JDBC, which has its own model of transaction control.

Examples

The following Embedded SQL example demonstrates how a **COMMIT** restores the transaction level counter (*$TLEVEL*) to the level immediately prior to the **START TRANSACTION**, regardless of how many **SAVEPOINTS** have been established within the transaction. Note that the second **START TRANSACTION** in this program is a no-op which has no effect on *$TLEVEL*:

ObjectScript

```
&sql(SET TRANSACTION %COMMITMODE EXPLICIT)
   WRITE !,"Set transaction mode, SQLCODE=",SQLCODE
   WRITE !,"Transaction level=",$TLEVEL
&sql(START TRANSACTION)
   WRITE !,"Start transaction, SQLCODE=",SQLCODE
   WRITE !,"Transaction level=",$TLEVEL
&sql(SAVEPOINT a)
   WRITE !,"Set Savepoint a, SQLCODE=",SQLCODE
   WRITE !,"Transaction level=",$TLEVEL
&sql(SAVEPOINT b)
   WRITE !,"Set Savepoint b, SQLCODE=",SQLCODE
   WRITE !,"Transaction level=",$TLEVEL
&sql(START TRANSACTION) /* Performs no operation */
   WRITE !,"Start transaction, SQLCODE=",SQLCODE
   WRITE !,"Transaction level=",$TLEVEL
&sql(SAVEPOINT c)
   WRITE !,"Set Savepoint c, SQLCODE=",SQLCODE
   WRITE !,"Transaction level=",$TLEVEL
&sql(COMMIT)
   WRITE !,"Commit transaction, SQLCODE=",SQLCODE
   WRITE !,"Transaction level=",$TLEVEL
```

The following Embedded SQL example demonstrates that the first **COMMIT** statement commits the entire transaction and that extra **COMMIT** statements have no effect and do not result in an error:

ObjectScript

```
&sql(SET TRANSACTION %COMMITMODE EXPLICIT)
   WRITE !,"Set transaction mode, SQLCODE=",SQLCODE
   WRITE !,"Transaction level=",$TLEVEL
&sql(START TRANSACTION)
   WRITE !,"Start transaction, SQLCODE=",SQLCODE
   WRITE !,"Transaction level=",$TLEVEL
&sql(SAVEPOINT a)
   WRITE !,"Set Savepoint a, SQLCODE=",SQLCODE
   WRITE !,"Transaction level=",$TLEVEL
&sql(COMMIT)
   WRITE !,"Commit transaction, SQLCODE=",SQLCODE
   WRITE !,"Transaction level=",$TLEVEL
&sql(COMMIT)   /* Performs no operation */
   WRITE !,"Commit again, SQLCODE=",SQLCODE
   WRITE !,"Transaction level=",$TLEVEL
&sql(COMMIT)   /* Performs no operation */
   WRITE !,"Commit again, SQLCODE=",SQLCODE
   WRITE !,"Transaction level=",$TLEVEL
```

See Also

- SQL commands: ROLLBACK SAVEPOINT SET TRANSACTION START TRANSACTION $TLEVEL

- Transaction Processing

- ObjectScript command: TCOMMIT

CREATE AGGREGATE (SQL)

Creates a user-defined aggregate function.

Synopsis

```
CREATE [OR REPLACE] AGGREGATE name(parameter_list) [ RETURNS datatype ]
    [ INITIALIZE WITH function-name ]
      ITERATE WITH function-name
    [ MERGE WITH function-name ]
    [ FINALIZE WITH function-name ]
```

Description

The **CREATE AGGREGATE** command creates a user-defined aggregate function (UDAF). When invoked, this user-defined aggregate function iterates through the row values and invokes one or more user-defined functions to compute an aggregate value. You can use **CREATE AGGREGATE** to provide aggregate operations not provided by the standard InterSystems IRIS SQL aggregate functions.

If you invoke **CREATE AGGREGATE** to create a UDAF that already exists, SQL issues an SQLCODE -428 error, with a %msg such as: `User Defined Aggregate Function SQLUser.MyUDAF already exists`. If you specify the optional OR REPLACE keyword clause (**CREATE OR REPLACE AGGREGATE**), specifying the name of an existing UDAF does not generate an error. Instead, the existing UDAF is updated with the specified definition.

To delete a user-defined aggregate function, use the DROP AGGREGATE command.

Privileges

The **CREATE AGGREGATE** command is a privileged operation. Before using **CREATE AGGREGATE** you must have Execute privilege for the UDAF and all referenced user-defined functions. Failing to do so results in an SQLCODE -99 error (Privilege Violation).

Aggregate Function Name

The UDAF *name* must be a valid identifier. Aggregate function names are not case-sensitive.

The UDAF *name* can be qualified (schema.aggname), or unqualified (aggname). An unqualified *name* takes the default schema name.

The UDAF *name* cannot be the same as the name of an existing stored procedure. Attempted to create a UDAF name that duplicates a stored procedure name generates an SQLCODE -428 error, with a %msg such as `User Defined Aggregate Function SQLUser.MyFunction conflicts with existing stored procedure name`.

INITIALIZE WITH Clause

The optional INITIALIZE WITH clause invokes the specified user-defined function or class method to compose the initial state object. The state object value is used to pass interim aggregate values or other variables required to perform the end calculation. If this clause is not specified, a null object is passed as the initial state object to the function specified in the ITERATE WITH clause.

The specified user-defined *function-name* must exist when **CREATE AGGREGATE** is invoked; otherwise an SQLCODE -428 error is generated and %msg specifies the UDAF function, the clause, and the non-existent function name.

The following is a user-defined function that defines an initial state object:

SQL

```
CREATE FUNCTION MyAggregateInit() returns varchar language ObjectScript { RETURN "^" }
```

ITERATE WITH Clause

The ITERATE WITH clause invokes the specified user-defined function or class method once for each row being aggregated. It take a state object representing the interim result and the current row's column value(s) as input parameters and performs its operation on that state object, which accumulates the aggregate value. When all rows have been processed it returns the new state value.

The specified user-defined *function-name* must exist when **CREATE AGGREGATE** is invoked; otherwise an SQLCODE -428 error is generated and %msg specifies the UDAF function, the clause, and the non-existent function name.

MERGE WITH Clause

The optional MERGE WITH clause can be specified to enable parallel processing of the user-defined aggregate function. If not specified, the query invoking the UDAF uses single-thread processing. For further details, see Parallel Processing.

FINALIZE WITH Clause

The optional FINALIZE WITH clause invokes the specified user-defined function or class method once, at the end of processing, to perform any final calculations based on the state value returned from the last call to the ITERATE WITH clause function. If the invoking query specifies a GROUP BY clause, this user-defined function is invoked once for each GROUP BY grouped value.

The specified user-defined *function-name* must exist when **CREATE AGGREGATE** is invoked; otherwise an SQLCODE -428 error is generated and %msg specifies the UDAF function, the clause, and the non-existent function name.

Arguments

name

The name of the user-defined aggregate function to be created. The *name* must be a valid identifier. The *name* can be qualified (schema.aggname), or unqualified (aggname). An unqualified *name* takes the default schema name. Aggregate function names are not case-sensitive. The *name* must be followed by parentheses containing one or more parameters.

parameter_list

A list of parameters used to pass values to the aggregate function. The parameter list is enclosed in parentheses. You can specify a single parameter, or a list of parameters separated by commas. Each parameter in the list consists of a parameter name and a data type. For example: (param1 INTEGER,param2 NUMERIC).

RETURNS datatype

An optional argument that specifies the data type to return the aggregate function value. If omitted, the data type defaults to the data type of the first parameter in the *parameter_list*.

function-name

The name of an existing user-defined function created using the CREATE FUNCTION command, or a class method that returns a value and is projected as an SQL procedure. A user-defined function is stored as a method in a stored procedure class. For example, the user-defined function MyFunction takes the default schema name: SQLUser.MyFunction, which corresponds to the class User.funcMyFunction which contains the classmethod MyFunction().

Invoking a User-defined Aggregate Function

User-defined aggregate functions follow the same usage rules as standard aggregate functions.

A UDAF is invoked in a SELECT list, either as a listed *select-item* or in a subquery *select-item*. It can specify a column alias; if a column alias is not specified, it defaults to Aggregate_n. For example,

SQL

```
SELECT Home_State,AVG(Age) AS AvgAge,MAX(Age) AS MaxAge,SecondHighest(Age) AS SecondMaxAge
FROM Sample.Person GROUP BY Home_State
```

A UDAF cannot be used directly in an ORDER BY clause. Attempting to do so generates an SQLCODE -73 error. However, you can use a user-defined aggregate function in an ORDER BY clause by specifying the corresponding column alias or *select-item* sequence number.

A UDAF can be used directly in a HAVING clause. However, a HAVING clause must explicitly specify the user-defined aggregate function; it cannot specify a UDAF using the corresponding *select-item* column alias or *select-item* sequence number.

An aggregate function *cannot* be used directly in:

- a WHERE clause. Attempting to do so generates an SQLCODE -19 error.

- a GROUP BY clause. Attempting to do so generates an SQLCODE -19 error.

- a TOP clause. Attempting to do so generates an SQLCODE -1 error.

- a JOIN. Attempting to specify an aggregate in an ON clause generates an SQLCODE -19 error. Attempting to specify an aggregate in a USING clause generates an SQLCODE -1 error.

Unlike a standard aggregate function, a user-defined aggregate function cannot specify a DISTINCT, %FOREACH, or %AFTERHAVING clause.

Parallel Processing

If the optional MERGE WITH clause is specified, the MERGE WITH function merges the supplied state objects coming from the ITERATE WITH functions of two or more parallel subqueries, returning a single merged values that represents the aggregated state. The MERGE WITH function is automatically invoked as many times as the number of parallel processes. The result of these merges is supplied to the FINALIZE WITH clause.

When declaring a MERGE WITH function, it is assumed the state object supports implicit serialization. For example, by implementing the %SerialObject interface in ObjectScript.

If a MERGE WITH function is not provided, the user-defined aggregate function is not processed by parallel threads when %PARALLEL or sharding is specified. It is porcessed as a single thread.

Listing User-defined Aggregate Functions

The INFORMATION.SCHEMA.USERDEFINEDAGGREGATES persistent class displays information about all user-defined aggregate functions in the current namespace. It provides a number of properties including the names of the user-defined functions specified in its clauses.

The following example returns the schema name, user-defined aggregate name, ITERATE clause function name, and returned data type for all user-defined aggregate functions in the current namespace:

SQL

```
SELECT AGGREGATE_SCHEMA,AGGREGATE_NAME,ITERATE_FUNCTION,RETURN_TYPE
FROM INFORMATION_SCHEMA.USER_DEFINED_AGGREGATES
```

If no RETURNS clause is specified, the RETURN_TYPE value is NULL.

Example

The following example creates a user-defined aggregate function that sums values by adding all high (>=5) values and subtracting 5 for all low (<5) values. All values are data type NUMERIC(4,1). The first step is to create the iterate function, specifying a state variable (*tot*) and an input variable (*num*):

SQL

```
CREATE FUNCTION Sample.AddSub(tot NUMERIC(4,1),IN num NUMERIC(4,1)) RETURNS NUMERIC(4,1)
LANGUAGE OBJECTSCRIPT {IF num>=5 {SET tot=tot+num} ELSE {SET tot=tot-5} QUIT tot}
```

You can then define the user-defined aggregate function:

```
CREATE AGGREGATE Sample.SumAddSub(arg NUMERIC(4,1))
   ITERATE WITH Sample.AddSub
```

You can then invoke this user-defined aggregate function for the Score field as follows:

```
SELECT TestSubject,Score,SUM(Score) AS ScoreSum,Sample.SumAddSub(Score) AS ScoreAddHighSubtractLow
```

To avoid negative values, add a FINALIZE WITH function:

```
CREATE FUNCTION Sample.NoNeg(tot NUMERIC(4,1)) RETURNS NUMERIC(4,1)
LANGUAGE OBJECTSCRIPT {IF num>0 {QUIT tot} ELSE {SET tot=0 QUIT tot}}

CREATE OR REPLACE AGGREGATE Sample.SumAddSub(arg NUMERIC(4,1))
   ITERATE WITH Sample.AddSub
   FINALIZE WITH Sample.NoNeg
```

See Also

- CREATE FUNCTION command

- DROP AGGREGATE command

- Overview of Aggregate Functions

- SQLCODE error messages

CREATE DATABASE (SQL)

Creates a database (namespace).

Synopsis

```
CREATE DATABASE dbname [ON DIRECTORY pathname]
    [WITH [ENCRYPTED_DB]
    [GLOBAL_JOURNAL_STATE [=] {YES | NO}] ]
```

Description

The **CREATE DATABASE** command creates a namespace and two associated databases. This allows you to create a namespace within SQL.

The specified *dbname* is the name of the created namespace and the directory that contains the corresponding database files. Namespace names are not case-sensitive. A *dbname* follows the naming conventions for an SQL identifier, with the following additional restrictions:

- An underscore (_) character is not permitted as the first character of *dbname* (but may be used elsewhere within the name). The @, #, and $ characters are not permitted in *dbname*. Attempting to include these invalid characters in *dbname* generates an SQLCODE -343 error.

- A hyphen (-) character is not permitted in *dbname* (hyphen is not a valid SQL identifier character). However, a namespace name created by other means can include a hyphen character.

- A *dbname* cannot be longer than 63 characters; specifying a longer *dbname* generates an SQLCODE -400 fatal error with the appropriate %msg.

If the specified *dbname* namespace already exists, InterSystems IRIS issues an SQLCODE -341 error.

You can specify neither, either, or both WITH options: ENCRYPTED_DB and/or GLOBAL_JOURNAL_STATE. If you specify both, they are separated by a space, as follows: WITH ENCRYPTED_DB GLOBAL_JOURNAL_STATE=NO.

By default, **CREATE DATABASE** creates two databases in the mgr directory with the *dbname* name subdirectory containing two subdirectories, C (code) and D (data). Each of these subdirectories contains a IRIS.DAT file, a iris.lck file, and an empty stream folder. For example, on a Windows system, CREATE DATABASE Barney would create the namespace BARNEY and the following database files:

```
c:\InterSystems\IRIS\mgr\Barney\C containing IRIS.DAT, iris.lck, stream folder
c:\InterSystems\IRIS\mgr\Barney\D containing IRIS.DAT, iris.lck, stream folder
```

The C (code) directory is used for the namespace routines database. The D (data) directory is used for the namespace globals database. To return the location of the mgr directory, use the **%SYSTEM.Util.ManagerDirectory()** method.

The optional ON DIRECTORY *pathname* clause allows you to specify a different location for the database files, rather than a directory with the same name as the namespace. For example:

SQL

```
CREATE DATABASE Flintstone ON DIRECTORY 'c:\InterSystems\IRIS\mgr\Fred'
```

If you specify a *pathname* that already exists, InterSystems IRIS issues an SQLCODE -341 error.

The **CREATE DATABASE** command is a privileged operation. Prior to using **CREATE DATABASE**, it is necessary to be logged in as a user with the %Admin_Manage resource. Failing to do so results in an SQLCODE -99 error (Privilege Violation).

Use the **$SYSTEM.Security.Login()** method to assign a user with appropriate privileges:

ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql(      )
```

You must have the **%Service_Login:Use** privilege to invoke the **$SYSTEM.Security.Login** method. For further information, see %SYSTEM.Security.

You can also create a namespace from the Management Portal. Select **System Administration**, **Configuration**, **System Configuration**, **Namespaces** to list the existing namespaces. At the top of this table of existing namespaces you can click **Create New Namespace**.

The maximum number of namespaces on a single InterSystems IRIS instance is 2048.

Arguments

dbname

The name of the database (namespace) to be created.

pathname

An optional argument that denotes the root pathname location for the databases, specified as a quoted string. The C and D directories are created as subdirectories of this root path. The default is to create the database in the mgr directory.

WITH ENCRYPTED_DB

An optional argument that specifies whether or not the database is encrypted. The default is not encrypted.

WITH GLOBAL_JOURNAL_STATE

An optional argument that specifies whether or not the database is journaled. YES specifies that the database is journaled (which is recommended). NO specifies that the database is not journaled. The equal sign (=) is optional. The default is journaled.

See Also

- DROP DATABASE command
- USE DATABASE command

CREATE FOREIGN SERVER (SQL)

Creates a foreign server.

Synopsis

```
CREATE [ FOREIGN ] SERVER server-name [ TYPE server-type ] FOREIGN DATA WRAPPER CSV
HOST host-name CREATE [ FOREIGN ] SERVER server-name [ TYPE server-type ] FOREIGN
DATA WRAPPER JDBC CONNECTION connection-name id-options
```

Arguments

Arguments	Description
server-name	The name for the foreign server definition being created. A valid identifier, subject to the same additional naming restrictions as a table name. A foreign server name is a qualified name.
TYPE *server-type*	The type of the foreign server. Foreign servers can be of two types: 'DB' or 'FILE'. Note that the delimiters are required.
FOREIGN DATA WRAPPER [CSV | JDBC]	Describes the protocol that will be used to access external data. The foreign data wrapper can be one of two options: CSV or JDBC.
HOST *host-name*	The source that stores the data. This name is a folder in a file system. The *host-name* must be delimited by single quotation marks.
CONNECTION *connection-name*	The name of the JDBC connection that connects InterSystems IRIS to the external system. Must be delimited. For details on establishing a JDBC connection, see Connecting the SQL Gateway via JDBC.
id-options	*Optional* — Either DELIMITEDIDS or NODELIMITEDIDS. Specifies whether the external data source accepts delimited identifiers or not.

Description

The CREATE FOREIGN SERVER command defines a remote location that InterSystems SQL can use to access an external data source, called a foreign server. This command stores metadata that the system can use to project data from an external data source into foreign tables that can be queried alongside native tables. In addition, it defines the foreign data wrapper, which determines the protocol that the foreign server uses to access data from an external source.

InterSystems SQL currently supports two types of foreign servers (optionally specified with the TYPE keyword), 'FILE' and 'DB', that retrieve external data from either a .csv file or database, respectively. Foreign servers of type 'FILE' access files in file systems, while foreign servers of type 'DB' use pre-defined JDBC connections to access external databases. The type of a foreign server is implicitly set by the foreign data wrapper.

Create a Foreign Server for .csv Files

When defining a foreign server that will create foreign tables by reading data stored in .csv files, you will use the CSV foreign data wrapper. Foreign servers defined in this manner must define, at minimum, a local file path that stores any .csv files that you may project into InterSystems IRIS. This file path is specified by using the HOST keyword.

The following example creates a foreign server that accesses .csv files:

```
CREATE FOREIGN SERVER Sample.DumpDir FOREIGN DATA WRAPPER CSV HOST '/data/dumps'
```

Create a Foreign Server with a JDBC Connection

When defining a foreign server that will create foreign tables by reading data stored in an external database, you will use the JDBC foreign data wrapper. Foreign servers defined in this manner must specify a JDBC connection that will connect the instance of InterSystems IRIS with the external data source. This connection's name is specified by using the CONNECTION keyword.

The following example creates a foreign server for JDBC connections:

```
CREATE FOREIGN SERVER Sample.Postgres FOREIGN DATA WRAPPER JDBC CONNECTION 'PostgresSQLConnection'
```

Using Delimited Identifiers

When connecting to an external data source, you may need to specify whether or not the foreign server should accept delimited identifiers. By default, InterSystems IRIS may send delimited identifiers to an external database management system when creating a projection, but not all database management systems allow delimited identifiers. If you are using an external database management system that does not accept delimited identifiers, you should specify the NODELIMITE-DIDS at the end of your **CREATE FOREIGN SERVER** command. The default setting allows delimited identifiers.

See Also

- DROP FOREIGN SERVER
- ALTER FOREIGN SERVER
- CREATE FOREIGN TABLE

CREATE FOREIGN TABLE (SQL)

Creates a foreign table.

Synopsis

Foreign Table from File

```
CREATE FOREIGN TABLE [ IF NOT EXISTS ] table-name ( column type, column2 type2, ... )      SERVER
server-name FILE file-name [ USING json-options ] CREATE FOREIGN TABLE [ IF NOT EXISTS ] table-name (
 column type, column2 type2, ... )              SERVER server-name FILE file-name COLUMNS ( col-name,
col-name2, ... ) [ USING json-options ] CREATE FOREIGN TABLE [ IF NOT EXISTS ] table-name ( column
type, column2 type2, ... )      SERVER server-name FILE file-name COLUMNS ( col-name, col-name2, ... )
 VALUES ( header, header2, ... ) [ USING json-options ] CREATE FOREIGN TABLE [ IF NOT EXISTS ] table-name
 ( column type, column2 type2, ... )      SERVER server-name FILE file-name VALUES ( header, header2,
... ) [ USING json-options ]
```

Foreign Table from Database

```
CREATE FOREIGN TABLE [ IF NOT EXISTS ] table-name [ ( column type, column2 type2, ... ) ] SERVER
server-name CREATE FOREIGN TABLE [ IF NOT EXISTS ] table-name [ ( column type, column2 type2, ... ) ]

  SERVER server-name TABLE external-table [ VALUES ( header, header2, ... ) ] CREATE FOREIGN TABLE [
IF NOT EXISTS ] table-name [ ( column type, column2 type2, ... ) ]      SERVER server-name QUERY query
 CREATE FOREIGN
TABLE [ IF NOT EXISTS ] table-name [ ( column type, column2 type2, ... ) ]      SERVER server-name VALUES
 ( header, header2,
... )
```

Description

The **CREATE FOREIGN TABLE** command creates a foreign table definition in the specified structure. **CREATE FOREIGN TABLE** creates a projection of data from an external data source than can be queried alongside data native to InterSystems IRIS.

If you do not specify the IF NOT EXISTS option and attempt to create a foreign table with the same name as a pre-existing foreign table, the system returns an SQLCODE -201 error. The IF NOT EXISTS option suppresses the error, but InterSystems IRIS does not recreate the foreign table.

Suppresses the error that arises if a schema with *name* already exists. The schema is not re-created.

If you create a foreign table from a .csv file, you may specify projection options by using a JSON object or a string containing a JSON object in a USING clause, just as you might with a LOAD DATA command.

Foreign Table from File

You can create a foreign table that projects data from a file external to your instance of InterSystems IRIS. In these cases, the foreign server on which you create the table must use CSV as its foreign data wrapper. Note that there is a slight difference in behavior between usage when the file does and does not.

- **CREATE FOREIGN TABLE [IF NOT EXISTS]** *table-name* **(** *column type*, *column2 type2*, **...) SERVER** *server-name* **FILE** *file-name* **[USING** *json-options* **]** creates a foreign table that projects data stored in the specified file name.

  - If the file does not have a header, the columns in the new foreign table contain data from the first *n* columns in the file, where *n* is the length of the primary column list. Within InterSystems SQL, you can query this foreign table by referring to the column names in the primary column list.

    ```
    CREATE FOREIGN TABLE (
       firstName VARCHAR(15),
       lastName VARCHAR(15),
       DOB DATE
    ) Sample.Person SERVER Sample.HospitalDir FILE 'person.csv'
    ```

– If the file does have a header, The column names in the primary column list must correspond with header names of columns in the file. Only the column names in the file that correspond to column names in the primary column list appear in the projected table.

```
CREATE FOREIGN TABLE (
    firstName VARCHAR(15),
    lastName VARCHAR(15),
    DOB DATE
) Sample.Person SERVER Sample.HospitalDir FILE 'person.csv' USING { "from": { "file": { "header":
true } } }
```

- **CREATE FOREIGN TABLE [IF NOT EXISTS]** *table-name* **(** *column type***,** *column2 type2***, ...) SERVER** *server-name* **FILE** *file-name* **COLUMNS (** *col-name type***,** *col-name2 type2***, ...) [USING** *json-options* **]** creates a foreign table that projects data stored from the specified file with a column order specified by the COLUMNS clause. Names in the primary column list specify the names of the columns in the table and positionally correlate with the columns of the file. The names in the COLUMNS clause must be identical to the names in the primary column list and each name must appear in both lists. The COLUMNS clause can be used to reorder the columns from the file; the order of columns in the COLUMNS clause does not need to match the order of the columns in the primary column list. The order of the columns in the foreign table is determined by the position of the column names in the COLUMNS clause.

  If the file has a header, this command behaves identically, as the file's header is disregarded. In this case, you should specify the `from.file.header` JSON option as true in the USING clause.

```
CREATE FOREIGN TABLE Sample.Person (
    FileColumnOne VARCHAR(10),
    FileColumnTwo VARCHAR(20)
) SERVER Sample.HospitalDir FILE person.csv COLUMNS (FileColumnTwo VARCHAR(20), FileColumnOne
VARCHAR(10))
```

- **CREATE FOREIGN TABLE [IF NOT EXISTS]** *table-name* **(** *column type***,** *column2 type2***, ...) SERVER** *server-name* **FILE** *file-name* **COLUMNS (** *col-name type***,** *col-name2 type2***, ...) VALUES (** *header***,** *header2***, ...) [USING** *json-options* **]** creates a foreign table that projects data stored from the specified file with a column order specified by the VALUES clause, possibly omitting certain columns from the .csv file. The primary column list defines the column names and types that appear in the foreign table. The COLUMNS clause lists the columns in the file and their type; the length of this list can be longer than the length of the primary column list and the names need be similar. The VALUES clause reorders the column names in the COLUMNS clause, but is the length of the primary columns list.

  You may use the VALUES clause to omit certain columns from the file (specified in the COLUMNS clause) from the foreign table. The order of names in the VALUES clause is mapped onto the order of column names in the primary column list. Within InterSystems SQL, you can query this foreign table by referring to the column names in the primary column list.

  If the file has a header, this command behaves identically, as the file's header is disregarded. In this case, you should specify the `from.file.header` JSON option as true in the USING clause.

  In the following example, the FieldOne column projects data from the second element of the COLUMNS clause, the FieldTwo column projects data from the first element of the COLUMNS clause, and the FieldThree column projects data from the fourth element of the COLUMNS clause.

```
CREATE FOREIGN TABLE Sample.Person (
    FieldOne VARCHAR(10),
    FieldTwo VARCHAR(20),
    FieldThree INTEGER
) SERVER Sample.HospitalDB FILE person.csv COLUMNS (FirstName VARCHAR(10), LastName(20), DOB DATE,
 Age INTEGER) VALUES (LastName, FirstName, Age)
```

- **CREATE FOREIGN TABLE [IF NOT EXISTS]** *table-name* **(** *column type***,** *column2 type2***, ...) SERVER** *server-name* **FILE** *file-name* **VALUES (** *header***,** *header2***, ...) [USING** *json-options* **]** creates a foreign table that projects a subset of data stored from the specified file into the table. The column names in the VALUES clause must correspond to column names in the *.csv* file, which may be different from the names in the primary column list. The

order of columns in the foreign table is determined by the order of the columns in the primary column list, with the data in those columns coming from the positionally related element of the VALUES clause.

If the file does not have a header, the VALUES clause is ignored and meaningless.

```
CREATE FOREIGN TABLE Sample.Person (
    FirstName VARCHAR(10),
    LastName VARCHAR(20)
) SERVER Sample.HospitalDB FILE person.csv VALUES (FirstNameInFile, LastNameInFile) USING { "from":
 { "file": { "header": 1 } } }
```

Foreign Table from Database

You can create a foreign table that projects data from a database external to your instance of InterSystems IRIS. In these cases, the foreign server on which you create the table must use JDBC as its foreign data wrapper.

- **CREATE FOREIGN TABLE [IF NOT EXISTS]** *table-name* **(** *column type*, *column2 type2*, ... **) SERVER** *server-name* **[TABLE** *external-table* **]** creates a foreign table that projects data from a table that exists in specified table. The created table has the same columns as the table in the external database. If you omit the TABLE clause, InterSystems IRIS attempts to access a table on the foreign server using *table-name*, rather than *external-table*.

```
CREATE FOREIGN TABLE Sample.Person (
    FirstName VARCHAR(10),
    LastName VARCHAR(20)
) SERVER Sample.ExternalDB TABLE 'hospital.people'
```

- **CREATE FOREIGN TABLE [IF NOT EXISTS]** *table-name* **(** *column type*, *column2 type2*, ... **) SERVER** *server-name* **QUERY** *query* creates a foreign table that projects data returned from executing a query, specified by *query*, against a table that exists in an external database. InterSystems SQL does not validate the query before attempting to execute it against the external database.

```
CREATE FOREIGN TABLE Sample.Team (
    FirstName VARCHAR(10),
    LastName VARCHAR(20)
) SERVER Sample.ExternalDB QUERY 'SELECT FirstName,LastName FROM Hospital.Patients'
```

- **CREATE FOREIGN TABLE [IF NOT EXISTS]** *table-name* **(** *column type*, *column2 type2*, ... **) SERVER** *server-name* **[TABLE** *external-table* **] VALUES (** *header*, *header2*, ... **)** creates a foreign table that projects data stored in the specified table with column names that differ from those in the external data source. If you omit the TABLE clause, InterSystems IRIS attempts to access a table on the foreign server using *table-name*, rather than *external-table*. The headers named in the VALUES clause identify the column names from the external data source, but may differ from the names that you have specified in the column list. Consequently, the VALUES clause must have the same number of columns as the column list.

```
CREATE FOREIGN TABLE Sample.Team (
    TeamID BIGINT,
    Name VARCHAR(100)
) SERVER Sample.ExternalDB TABLE 'hospital.teams' VALUES (team_id, name)
```

Arguments

table-name

In a **CREATE FOREIGN TABLE** command, this argument specifies the name of the foreign table that you want to create as a valid identifier. A table name can be qualified or unqualified.

- An unqualified foreign table name has the following syntax: **tablename**; it omits schema (and the period (.) character). An unqualified table name takes the default schema name. The initial system-wide default schema name is SQLUser, which corresponds to the default class package name User. Schema search path values are ignored.

If you have created a foreign table using a JDBC connection and omitted the TABLE clause, then the unqualified table name is leveraged against the external data source to create the project, but the table is accessible through InterSystems SQL under the default schema qualified name.

If you have specified an unqualified foreign table name with a JDBC connection and do not specify a TABLE clause, then the

The system-wide default schema name can be configured.

To determine the current system-wide default schema name, use the **$SYSTEM.SQL.Schema.Default()** method.

- A qualified foreign table name has the following syntax: **schema.tablename**. It can specify either an existing schema name or a new schema name. Specifying an existing schema name places the foreign table within that schema. Specifying a new schema name creates a new schema and associated class package, and places the table within that schema.

column

In a CREATE FOREIGN TABLE command, specify the column name or a comma-separated list of column names, used to define the columns of the table you are creating, in the primary column list. You can specify the column names in any order, with a space separating the column name from its associate data type. By convention, each column definition is usually presented on a separate line and indentation is used. This convention is recommended for readability, but is not required.

Enclose primary column lists in parentheses.

type

The InterSystems SQL data type class of the column name specified by column. A specified data type limits a column's allowed data values to the values appropriate for that data type. InterSystems SQL supports most standard SQL data types.

Data from the external data source is coerced into the specified type as part of the project. If the field cannot be coerced, such as an invalid date format, a runtime error is raised.

server-name

In a **CREATE FOREIGN TABLE** command, this argument specifies the foreign server configuration that accesses the external data source.

You may specify a qualified or unqualified foreign server name. If you specify an unqualified foreign server name, the system attempts to locate the foreign server within the default schema, which is SQLUser by default. If you specify a qualified foreign server name, the system attempts to locate the foreign server within the provided schema.

If the foreign server cannot be located within the determined schema, the system raises an SQLCODE -360 error.

file-name

In a **CREATE FOREIGN TABLE** command, this argument specifies the location of a .csv file containing the data to project into InterSystems IRIS, defined as a complete file path enclosed in quotes. This argument should only be used when the foreign server specified in the command uses the CSV option for its foreign data wrapper.

- Each line in a file specifies a separate row to be projected into the foreign table. Newline ("\n") is the default line separator. Blank lines are ignored.

- Data values in a row are separated by a column separator character. A comma is the default column separator character. All data fields must be indicated by column separators, including unspecified data indicated by placeholder column separators. You can define a different column separator character by specifying the **columnseparator** option in the **USING** *json-options* clause.

- By default, no escape character is defined. To include the column separator character as a literal in a data value, enclose the data value in quotation marks. To include a quotation mark in a quoted data value, double the quote character. You can define an escape character specifying the **escapechar** option in the **USING** *json-options* clause.

- By default, data values are specified in the order of the fields in the foreign table. You can use the COLUMNS and VALUES clauses to specify the data in a different order.

- All data in a .csv file is validated against the table's data criteria, including the number of data fields in the record, and the data type and data length for each field. If a certain record in the file cannot be validated, an error message is issued. Note that date and time constructs in .csv files must be in ODBC format, as other formats may produce errors or incorrect query results.

col-name

In a CREATE FOREIGN TABLE command, this argument appears in a COLUMNS clause. When the file has no header, the COLUMNS clause provides a name for the columns in the file. When the file has a header, the COLUMNS clause can often be omitted.

header

In a CREATE FOREIGN TABLE command, this argument appears in a VALUES clause. A VALUES clause may be used in a variety of scenarios

external-table

In a CREATE FOREIGN TABLE command that connects to an eternal data source through a JDBC connections, this argument supplies the name of the external table to project into InterSystems IRIS. If you omitted a column list, a foreign table created in this manner copies the column definitions, including column names and data types (where supported), from the data source.

query

In a CREATE FOREIGN TABLE command that connects to an external data source through a JDBC connection, this argument supplies the column definitions and column data for a foreign table by querying a table in the external data source. It is a SELECT query that is executed against the external data source.

Foreign tables created in this way copy column definitions from the external data source, including column names and data types (when supported). A foreign table can copy column definitions from multiple tables if the query specifies joined tables from the external data source.

json-options

This argument specifies loading options as a JSON object or a string containing a JSON object in the USING clause. Its usage is nearly identical to the corresponding argument in the LOAD DATA command. For a complete overview on the syntax and options, refer to the LOAD DATA documentation. Note that the CREATE FOREIGN TABLE command supports only the options in the `from.file` tree.

See Also

- CREATE FOREIGN SERVER

- ALTER FOREIGN TABLE

- DROP FOREIGN TABLE

- DROP FOREIGN SERVER

- LOAD DATA

CREATE FUNCTION (SQL)

Creates a function as a method in a class.

Synopsis

```
CREATE FUNCTION name(parameter_list) [characteristics]
     [ LANGUAGE SQL ]
    BEGIN code_body ;
    END

CREATE FUNCTION name(parameter_list) [characteristics]
     LANGUAGE OBJECTSCRIPT      { code_body }

CREATE FUNCTION name(parameter_list) [characteristics]
      LANGUAGE { JAVA | PYTHON | DOTNET } EXTERNAL NAME external-stored-procedure
```

Description

The **CREATE FUNCTION** statement creates a function as a method in a class. This class method is projected as an SQL Stored Procedure. You can also use the CREATE PROCEDURE statement to create a method which is projected as an SQL Stored Procedure. **CREATE FUNCTION** should be used when the method is to return a value, but it can be used to create a method that does not return a value.

The optional keyword **OR REPLACE** allows you to modify or replace an existing function. **CREATE OR REPLACE FUNCTION** has the same effect as invoking **DROP FUNCTION** to delete the old version of the function and then invoking **CREATE TRIGGER**.

In order to create a function, you must have %CREATE_FUNCTION administrative privilege, as specified by the GRANT command.

You cannot create a function in a class if the class definition is a deployed class. This operation fails with an SQLCODE -400 error alongside the %msg Unable to execute DDL that modifies a deployed class: 'classname'.

For information on calling SQL functions from within SQL statements, refer to User-defined Functions. For calling SQL stored procedures in a variety of contexts, refer to the CALL statement.

Arguments

name

The name of the function to be created in a stored procedure class. The *name* must be a valid identifier and must be followed by parentheses, even if no parameters are specified. This name may be unqualified (StoreName) and take the default schema name, or qualified by specifying the schema name (Patient.StoreName). You can use the **$SYSTEM.SQL.Schema.Default()** method to determine the current system-wide default schema name. The initial system-wide default schema name is SQLUser, which corresponds to the class package name User.

Note that the FOR characteristic (described below) overrides the class name specified in *name*. If a function with this name already exists, the operation fails with an SQLCODE -361 error.

The name of the generated class is the package name corresponding to the schema name, followed by a dot, "func", and then the specified *name*. For example, if the unqualified function name RandomLetter takes the initial default schema SQLUser, the resulting class name would be: User.funcRandomLetter. For further details, see SQL to Class Name Transformations.

InterSystems SQL does not allow you to specify a duplicate function name that differs only in letter case. Specifying a function name that differs only in letter case from an existing function name results in an SQLCODE -400 error.

parameter-list

An optional list of parameters used to pass values to the function. The parameter list is enclosed in parentheses, which are mandatory even when no parameters are specified, and parameter declarations in the list are separated by commas. Each parameter declaration in the list consists of (in order):

- An optional keyword specifying whether the parameter mode is IN (input value), OUT (output value), or INOUT (modify value). If omitted, the default parameter mode is IN.

- The parameter name. Parameter names are case-sensitive.

- The data type of the parameter.

- *Optional*: A default value for the parameter. You can specify the DEFAULT keyword followed by a default value; the DEFAULT keyword is optional. If no default is specified, the assumed default is NULL.

The following example specifies two input parameters, both of which have default values. The optional DEFAULT keyword is specified for the first parameter, omitted for the second parameter:

SQL

```
CREATE FUNCTION RandomLetter(IN firstlet CHAR DEFAULT 'A',IN lastlet CHAR 'Z')
BEGIN
-- SQL program code
END
```

User-defined functions are supplied to the clauses of a user-defined aggregate function. When defining a function for use in a user-defined aggregate function, you define a state parameter which is used to aggregate and pass the output value.

A function is "correlated" if it takes at least one parameter that is dependent on a value from a row of data, for example the %ID field. Correlated functions are evaluated per row; uncorrelated functions (that is, functions that either take no parameters or take arguments that remain consistent across all rows) are evaluated a single time.

characteristics

An optional argument that consists of one or more keywords specifying the characteristics of the function. Multiple characteristics are separated by whitespace (a space or line break), and characteristics can be specified in any order. The available keywords are as follows:

FOR *className*	Specifies the name of the class in which to create the function. If the class does not exist, it will be created. You can also specify a class name by qualifying the function name. The class name specified in the FOR clause overrides a class name specified by qualifying the function name.
FINAL	Specifies that subclasses cannot override the function. By default, functions are not final. The FINAL keyword is inherited by subclasses.
PRIVATE	Specifies that the function can only be invoked by other function of its own class or subclasses. By default, a function is public, and can be invoked without restriction. This restriction is inherited by subclasses.
PROCEDURE	Specifies that the function is projected as an SQL stored procedure. Stored procedures are inherited by subclasses. Because CREATE FUNCTION always projects an SQL stored procedure, this keyword is optional. This keyword can be abbreviated as PROC.
RETURNS *datatype*	Specifies the data type of the value returned by a call to the function. If RETURNS is omitted, the function cannot return a value. This specification is inherited by subclasses, and can be modified by subclasses. This *datatype* can specify type parameters such as MINVAL, MAXVAL, and SCALE. For example `RETURNS DECIMAL(19,4)`. Note that when returning a value, InterSystems IRIS ignores the length of *datatype*; for example, `RETURNS VARCHAR(32)` can receive a string of any length that is returned by a call to the function.
SELECTMODE *mode*	Only used when LANGUAGE is SQL (the default). When specified, InterSystems IRIS adds an #SQLCOMPILE SELECT=*mode* statement to the corresponding class method, thus generating the SQL statements defined in the method with the specified SELECTMODE. The possible *mode* values are LOGICAL, ODBC, RUNTIME, and DISPLAY. The default is LOGICAL.

The SELECTMODE clause is used for **SELECT** query operations and for **INSERT** and **UPDATE** operations. It specifies the compile-time select mode. The value that you specify for SELECTMODE is added at the beginning of the ObjectScript class method code as: #sqlcompile select=*mode*. For further details, see #sqlcompile select.

- In a **SELECT** query, the SELECTMODE specifies the mode in which data is returned. If the *mode* value is LOGICAL, then logical (internal storage) values are returned. For example, dates are returned in $HOROLOG format. If the *mode* value is ODBC, logical-to-ODBC conversion is applied, and ODBC format values are returned. If the *mode* value is DISPLAY, logical-to-display conversion is applied, and display format values are returned. If the *mode* value is RUNTIME, the display mode can be set (to LOGICAL, ODBC, or DISPLAY) at execution time.

- In an INSERT or UPDATE operation, the SELECTMODE RUNTIME option supports automatic conversion of input data values from a display format (DISPLAY or ODBC) to logical storage format. This compiled display-to-logical data conversion code is applied only if the select mode setting when the SQL code is executed is LOGICAL (which is the default for all InterSystems SQL execution interfaces).

When the SQL code is executed, the %SQL.Statement class *%SelectMode* property specifies the execution-time select mode, as described in Using Dynamic SQL. For further details on SelectMode options, refer to Data Display Options.

LANGUAGE

An optional keyword clause specifying the procedure code language. Available options are:

- LANGUAGE OBJECTSCRIPT (for ObjectScript) or LANGUAGE SQL. The procedure code is specified in the *code_body*.

- LANGUAGE JAVA, LANGUAGE PYTHON, or LANGUAGE DOTNET for an SQL procedure that invokes an external stored procedure in one of these languages. The syntax for an external stored procedure is as follows:

  ```
  LANGUAGE langname EXTERNAL NAME external-routine-name
  ```

  Where *langname* is JAVA, PYTHON, or DOTNET and *external-routine-name* is a quoted string containing the name of an external routine in the specified language. The SQL procedure invokes an existing routine; you cannot write code in these languages within the **CREATE FUNCTION** statement. Stored procedure libraries in these languages are stored external to IRIS, and therefore do not have to be packaged, imported, or compiled within IRIS. The following is an example of a **CREATE FUNCTION** that invokes an existing JAVA external stored procedure that returns a value:

  ```
  CREATE FUNCTION getPrice (item_name VARCHAR)
  RETURNS INTEGER
  LANGUAGE JAVA
  EXTERNAL NAME 'Orders.getPrice'
  ```

If the LANGUAGE clause is omitted, SQL is the default.

code_body

The program code for the method to be created. You specify this code in either SQL or ObjectScript. SQL program code is prefaced with a BEGIN keyword and concludes with an END keyword. Each complete SQL statement within *code_body* end with a semicolon (;). ObjectScript program code is enclosed in curly braces, and code lines must be indented. The language used must match the LANGUAGE clause. However, code specified in ObjectScript can contain embedded SQL.

InterSystems IRIS uses the code you supply to generate the actual code of the method. If the code you specify is SQL, InterSystems IRIS provides additional lines of code when generating the method that embed the SQL in an ObjectScript "wrapper," provide a procedure context handler (if necessary), and handle return values. The following is an example of this InterSystems IRIS-generated wrapper code:

ObjectScript

```
NEW SQLCODE,%ROWID,%ROWCOUNT,title
&sql( SELECT col FROM tbl )
QUIT $GET(title)
```

If the code you specify is OBJECTSCRIPT, the ObjectScript code must be enclosed in curly braces. All code lines must be indented from column 1, except for labels and macro preprocessor directives. A label or macro directive must be prefaced by a colon (:) in column 1.

For ObjectScript code, you must explicitly define the "wrapper" (which NEWs variables, and uses QUIT to exit and (optionally) to return a value upon completion).

When a stored procedure is called, an object of the class %Library.SQLProcContext is instantiated in the %sqlcontext variable. This procedure context handler is used to pass the procedure context back and forth between the procedure and its caller (for example, the ODBC server).

%sqlcontext consists of several properties, including an Error object, the SQLCODE error status, the SQL row count, and an error message. The following example shows the values used to set several of these:

```
SET %sqlcontext.%SQLCODE=SQLCODE
SET %sqlcontext.%ROWCOUNT=%ROWCOUNT
SET %sqlcontext.%Message=%msg
```

The values of SQLCODE and %ROWCOUNT are automatically set by the execution of an SQL statement. The %sqlcontext object is reset before each execution.

Alternatively, an error context can be established by instantiating a %SYSTEM.Error object and setting it as %sqlcontext.Error.

An SQLCODE -361 error is generated if the specified function already exists. To avoid this error, use the optional **OR REPLACE** keyword, or drop the old function first with **DROP FUNCTION**.

Executing a User-defined Function

You can execute a function in a **SELECT** statement, such as the following:

SQL

```
SELECT StudentName,StudentAge,SQLUser.HalfAge() AS HalfTheAge
FROM SQLUser.MyStudents
```

An SQLCODE -359 error is generated if the function does not exist.

An SQLCODE -149 error is generated if the execution of the function results in a error. The type of error is described in %msg.

Examples

The following example creates the RandomLetter() function (method) stored as a procedure that generates a random capital letter. You can then invoke this function in a **SELECT** statement. A **DROP FUNCTION** is provided to delete the RandomLetter() function. Note that this example is of an uncorrelated function, so the result set of the **SELECT** statement will contain Names that all start with the same, randomly chosen letter and will contain the number of names that start with the randomly chosen letter. An example result set is provided.

SQL

```
CREATE FUNCTION RandomLetter()
RETURNS INTEGER
PROCEDURE
LANGUAGE OBJECTSCRIPT
{
:Top
 SET x=$RANDOM(90)
 IF x<65 {GOTO Top}
 ELSE {QUIT $CHAR(x)}
}
```

SQL

```
SELECT Name FROM Sample.Person
WHERE Name %STARTSWITH RandomLetter()

Abbott, Amelia P.
Adams,John J.
Alton,Lionel N.
Amblin,Stephen O.
Amory,Jennifer E.
Andrews,Olivia G.
Arias,Rowan K.
Avery,Marvin N.

DROP FUNCTION RandomLetter
```

The following example creates the RandomLetter() function (method) stored as a procedure that generates a random capital letter as a correlated function that depends on the changing value of %ID, though the argument itself is not used within the

body of RandomLetter(). The result set of the **SELECT** statement will contain Names that start with different, randomly chosen letters and its length will contain a variable number of elements. An example result set is provided.

```
CREATE FUNCTION RandomLetter(IN id INTEGER)
RETURNS INTEGER
PROCEDURE
LANGUAGE OBJECTSCRIPT
{
:Top
 SET x=$RANDOM(90)
 IF x<65 {GOTO Top}
 ELSE {QUIT $CHAR(x)}
}

SELECT Name FROM Sample.Person
WHERE Name %STARTSWITH RandomLetter(%ID)

Alton,Lionel N.
Cooper,Peter H.
Hertz,Lana C.
Jones,Alyssa D.
```

The following example creates a function that invokes ObjectScript code, which in turn contains embedded SQL:

ObjectScript

```
 &sql(CREATE FUNCTION TraineeName(
  SSN VARCHAR(11),
  OUT Name VARCHAR(50) )
  PROCEDURE
   RETURNS VARCHAR(30)
   FOR SQLUser.MyStudents
   LANGUAGE OBJECTSCRIPT
   {
       NEW SQLCODE,%ROWCOUNT
       SET Name=""
       &sql(SELECT Name INTO :Name FROM Sample.Employee
           WHERE SSN = :SSN)
       IF $GET(%sqlcontext)'= "" {
         SET %sqlcontext.%SQLCODE=SQLCODE
         SET %sqlcontext.%ROWCOUNT=%ROWCOUNT }
         QUIT Name
   })
 IF SQLCODE=0 { WRITE !,"Created a function" QUIT}
 ELSE { WRITE !,"CREATE FUNCTION error: ",SQLCODE," ",%msg,!
   &sql(DROP FUNCTION TraineeName FROM SQLUser.MyStudents) }
   IF SQLCODE=0 { WRITE !,"Dropped a function" QUIT}
   ELSE { WRITE !,"Drop error: ",SQLCODE }
```

It uses the %sqlcontext object, and sets its %SQLCODE and %ROWCOUNT properties using the corresponding SQL variables. Note the curly braces enclosing the ObjectScript code following the function's LANGUAGE OBJECTSCRIPT keyword. Within the ObjectScript code there is Embedded SQL code, marked by `&sql` and enclosed in parentheses.

Security and Privileges

The CREATE FUNCTION command is a privileged operation that requires the user to have %Development:USE permission. Such permissions can be granted through the Management Portal. Executing a CREATE FUNCTION command without these privileges will result in an SQLCODE -99 error and the command will fail.

Users without proper permissions can still execute this command under one of two conditions:

• The command is executed via Embedded SQL, which does not perform privilege checks.

• The user explicitly specifies no privilege checking by, for example, calling either **%Prepare()** with the checkPriv argument set to 0 or **%ExecDirectNoPriv()** on a %SQL.Statement.

See Also

• DROP FUNCTION command

- **CREATE AGGREGATE** command

- Defining and Using Stored Procedures

CREATE INDEX (SQL)

Creates an index for a table.

Synopsis

```
CREATE [UNIQUE | BITMAP | BITMAPEXTENT | BITSLICE | COLUMNAR]
    INDEX index-name
    ON [TABLE] table-name (field-name, ...)
    [AS index-class-name [ (parameter-name = parameter_value, ... ) ] ]
    [WITH DATA  (datafield-name, ...)]
    [ [ IMMEDIATE | DEFER ] [BUILD] ]
```

Arguments

index-type

An optional argument that specifies the type of index to be created. The following are the options for the index type:

- UNIQUE: A constraint that ensures there will not be two rows in the table with identical values in all the fields in the index. You cannot specify this keyword for a bitmap or bitslice index.

  The UNIQUE keyword can be followed by (or replaced by) the CLUSTERED or NONCLUSTERED keywords. These keywords are no-ops; they are provided for compatibility with other vendors.

- BITMAP: Indicates that a bitmap index should be created. A bitmap index enables rapid queries on fields with a small number of distinct values.

- BITMAPEXTENT: Indicates that a bitmapextent index should be created. At most one bitmapextent index can be created for a table. No *field-name* is specified with BITMAPEXTENT.

- BITSLICE: Indicates that a bitslice index should be created. A bitslice index enables very fast evaluation of certain expressions, such as sums and range conditions. This is a specialized index type, which should only be used to solve very specific problems.

- COLUMNAR: Indicates that a columnar index should be created. A columnar index enables very fast queries, especially ones involving filtering and aggregation operations, on columns whose underlying data is stored across rows. Columnar indexes are an experimental feature for 2022.2.

index-name

The index being defined. The name is an identifier.

table-name

The name of an existing table for which the index is being defined. You cannot create an index for a view. A *table-name* can be qualified (schema.table), or unqualified (table). An unqualified table name takes the default schema name.

field-name

One or more field names that serve as the basis for the index. Field names must be enclosed in parentheses. Multiple field names are separated by commas.

Each field name can be followed by an ASC or DESC keyword. These keywords are no-ops; they are provided for compatibility with other vendors.

AS index-class-name

An optional argument specifying a class that defines an index, optionally followed by parentheses enclosing one or more comma-separated pairs of parameter names and associated values.

WITH DATA (datafield-name)

An optional argument that specifies one or more field names to be defined as Data properties for the index. Field names must be enclosed in parentheses. Multiple field names are separated by commas. You cannot specify a WITH DATA clause when specifying a BITMAP or BITSLICE index.

IMMEDIATE BUILD

An optional argument that specifies to build the index as soon as you create it. Indexes build immediately by default, so this clause can be omitted. The BUILD keyword is optional.

DEFER BUILD

An optional argument that specifies to disable building the index upon creation. This option also marks the index as not selectable, making it unavailable for use in queries. To later use the index, you must build it using **BUILD INDEX** and then make it selectable by using the **SetMapSelectability()** method; you can view whether a map is selectable or not in the Management Portal by navigating to **System Explorer** > **SQL** > **Catalog Details** and selecting the **Maps/Indices** button. The BUILD keyword is optional.

See additional compatibility syntax below.

Description

CREATE INDEX creates a sorted index on the specified field (or fields) of the named table. InterSystems IRIS uses indexes to improve performance of query operations. InterSystems IRIS automatically maintains indexes during **INSERT**, **UPDATE**, and **DELETE** operations, and this index maintenance may negatively affect performance of these data modification operations.

You can create an index using the **CREATE INDEX** command or by adding an index definition to a class definition, as described in Defining and Building Indexes. You can delete an index by using the DROP INDEX command.

For information about properties on which you can and cannot create indexes, see Properties That Can Be Indexed.

CREATE INDEX can be used to create any of the following types of index:

- A regular index (Type=index): Specify either **CREATE INDEX** (for non-unique values) or **CREATE UNIQUE INDEX** (for unique values).

- A bitmap index (Type=bitmap): Specify **CREATE BITMAP INDEX**.

- A bitslice index (Type=bitslice): Specify **CREATE BITSLICE INDEX**.

- A columnar index (Type=columnar): Specify **CREATE COLUMNAR INDEX**.

You can also define an index using the %Dictionary.IndexDefinition class.

You can use **CREATE INDEX** to add an index to a sharded table.

For information about indexes at the class level, see %Library.FunctionalIndex.

Privileges and Locking

The **CREATE INDEX** command is a privileged operation. The user must have %ALTER_TABLE administrative privilege to execute **CREATE INDEX**. Failing to do so results in an SQLCODE –99 error with the %msg `User 'name' does not have %ALTER_TABLE privileges`. You can use the GRANT command to assign %ALTER_TABLE privileges to a user or role, if you hold appropriate granting privileges. Administrative privileges are namespace-specific. For further details, refer to Privileges.

The user must have %ALTER privilege on the specified table. If the user is the Owner (creator) of the table, the user is automatically granted %ALTER privilege for that table. Otherwise, the user must be granted %ALTER privilege for the table. Failing to do so results in an SQLCODE –99 error with the %msg `User 'name' does not have required`

`%ALTER privilege needed to change the table definition for 'Schema.TableName'.` You can determine if the current user has %ALTER privilege by invoking the %CHECKPRIV command. You can use the GRANT command to assign %ALTER privilege to a specified table. For further details, refer to Privileges.

- **CREATE INDEX** cannot be used on a table projected from a persistent class, unless the table class definition includes [DdlAllowed]. Otherwise, the operation fails with an SQLCODE -300 error with the %msg `DDL not enabled for class 'Schema.tablename'.`

- **CREATE INDEX** cannot be used on a table projected from a deployed persistent class. This operation fails with an SQLCODE -400 error with the %msg `Unable to execute DDL that modifies a deployed class: 'classname'.`

The **CREATE INDEX** statement acquires a table-level lock on *table-name*. This prevents other processes from modifying the table's data. This lock is automatically released at the conclusion of the **CREATE INDEX** operation. **CREATE INDEX** maintains a lock on the corresponding class definition until the completion of the create index operation, including the population of the index data.

To create an index, the table cannot be locked by another process in either EXCLUSIVE MODE or SHARE MODE. Attempting a **CREATE INDEX** operation on a locked table results in an SQLCODE -110 error, with a %msg such as the following: `Unable to acquire exclusive table lock for table 'Sample.MyTest'.`

Options Supported for Compatibility Only

InterSystems SQL accepts the following **CREATE INDEX** options for parsing purposes only, to aid in the conversion of existing SQL code to InterSystems SQL. These options do not provide any actual functionality.

```
CLUSTERED | NONCLUSTERED owner.catalog. ASC | DESC
```

The following is an example showing the placement of these no-op keywords:

```
CREATE UNIQUE CLUSTERED INDEX index-name    ON TABLE owner.catalog.schema.table
   (field1 ASC, field2 DESC)
```

Index Name

The name of an index must be unique within a given table. Index names follow identifier conventions, subject to the restrictions below. By default, index names are simple identifiers; an index name can be a delimited identifier. An index name should not exceed 128 characters. Index names are not case-sensitive.

InterSystems IRIS uses the name you supply (which it refers to as the "SqlName") to generate a corresponding index property name in the class and the global. This index property name contains only alphanumeric characters (letters and numbers) and is a maximum of 96 characters in length. To generate an index property name, InterSystems IRIS first strips punctuation characters from the SqlName you supply, and then generates a unique identifier of 96 (or less) characters to create a unique index property name.

- An index name can be the same as a field, table, or view name, but such name duplication is not advised.

- An index property name (after punctuation stripping) must be unique. If you specify a duplicate SQL index name, the system generates an SQLCODE -324 error. If you specify an SQL index name that differs only in punctuation characters from an existing SQL index name, InterSystems IRIS substitutes a capital letter (beginning with "A") for the final character to create a unique index property name. Therefore it is possible (though not advisable) to create SQL index names that differ only in their punctuation characters.

- An index property name must begin with a letter. Therefore, either the first character of the index name or the first character after initial punctuation characters are stripped must be a letter. A valid letter is a character that passes the $ZNAME test. If the first character of the SQL index name is a punctuation character (% or _) and the second character is a number, InterSystems IRIS appends a lowercase "n" as the first character of the stripped index property name.

- An index name may be much longer than 31 characters, but index names that differ in their first 31 alphanumeric characters are much easier to work with.

The Management Portal SQL interface Catalog Details displays the SQL index name (**SQL Map Name**) and the corresponding index property name (**Index Name**) for each index.

What happens when you try to create an index with the same name as an existing index is described below.

Existing Index

By default, InterSystems IRIS rejects an attempt to create an index that has the same name as an existing index for that table and issues an SQLCODE -324 error. To determine the current setting, call **$SYSTEM.SQL.CurrentSettings()**, which displays a `Allow DDL CREATE INDEX for existing index` setting. The deault is 0, which is the recommended setting for this option. If this option is set to 1, InterSystems IRIS deletes the existing index from the class definition and then recreates it by performing the **CREATE INDEX**. It deletes the named index from the table specified in **CREATE INDEX**. This option permits the delete/recreate of a UNIQUE constraint index (which cannot be done using a **DROP INDEX** command). To delete/recreate a primary key index, refer to the ALTER TABLE command.

From the Management Portal, **System Administration**, **Configuration**, **SQL and Object Settings**, **SQL** you can set this option (and other similar create, alter, and drop options) system-wide by selecting the **Ignore redundant DDL statements** check box.

However, even if this option is set to allow the recreating of an existing index, you cannot recreate a Primary Key IDKEY index if the table contains data. Attempting to do so generates an SQLCODE -324 error.

Table Name

You must specify the name of an existing table.

- If *table-name* is a nonexistent table, **CREATE INDEX** fails with an SQLCODE -30 error, and sets %msg to `Table 'SQLUSER.MYTABLE' does not exist.`

- If *table-name* is a view, **CREATE INDEX** fails with an SQLCODE -30 error, and sets %msg to `Attempt to CREATE INDEX 'My_Index' on view SQLUSER.MYVIEW failed. Indices only supported for tables, not views.`.

Creating an index modifies the table's definition; if you do not have permission to change the table definition, **CREATE INDEX** fails with an SQLCODE -300 error, and sets %msg to `DDL not enabled for class 'schema.tablename'.`

Field Names

You must specify at least one field name to index on. Specify a field name or a comma-separated list of field names enclosed in parentheses. Duplicate field names are permitted and preserved in the index definition. Specifying more than one field may improve performance of **GROUP BY** operations, for example, group by state and then by city within each state. Generally, you should avoid indexing on a field or fields that have large amounts of duplicate data. For example, in a database of people, indexing on a Name field would be appropriate because most names are unique. Indexing on a State field would (in most cases) not be appropriate because of the large number of duplicate data values. The fields you specify must either be defined in the table or in the superclass of the table's persistent class. (all classes must, of course, have been compiled.) Specifying a nonexistent field generates an SQLCODE -31 error.

In addition to ordinary data fields, you can use **CREATE INDEX** to create an index:

- On a SERIAL field (a %Counter field).

- On an IDENTITY field.

- On the ELEMENTS or KEYS value for a collection.

You cannot create an index on a stream value field.

You cannot create an index with multiple IDKEY fields if one of the IDKEY fields (properties) is SQL Computed. This limitation does not apply to a single field IDKEY index. Because multiple IDKEY fields in an index are delimited using the "||" (double vertical bar) characters, you cannot include this character string in IDKEY field data.

Field in an Embedded Object (%SerialObject)

To index a field in an embedded object, you create an index in the table (%Persistent class) referencing that embedded object. In **CREATE INDEX** the *field-name* specifies the name of the referencing field in the table (%Persistent object) joined by an underbar to the field name in the embedded object (%SerialObject), as shown in the following example:

SQL

```
CREATE INDEX StateIdx ON TABLE Sample.Person (Home_State)
```

Here *Home* is a field in Sample.Person that references the embedded object Sample.Address, which contains the *State* field.

Only those embedded object records associated with the persistent class referencing property are indexed. You cannot index a %SerialObject property directly.

For further details on defining embedded objects (also known as serial objects) refer to Embedded Object (%SerialObject); for further details on indexing a property (field) defined in an embedded object, refer to Indexing an Embedded Object (%SerialObject) Property.

Index Class Name

This optional syntax allow users to specify a class and parameters for a functional index using SQL.

An SQL example is:

```
CREATE INDEX HistIdx ON TABLE Sample.Person (MedicalHistory) AS %iFind.Index.Basic (LANGUAGE='en',
LOWER=1)
```

For further details, refer to Indexing Sources for SQL Search.

WITH DATA Clause

Specifying this clause may allow a query to be resolved by only reading the index, which greatly reduces the amount of disk I/O, improving performance.

You should specify the same field in the *field-name* and the WITH DATA *datafield-name* if *field-name* uses string collation; this allows retrieval of the uncollated value without having to go to the Master Map. If the value in *field-name* does not use string collation there is no advantage to specifying this field in the WITH DATA *datafield-name*.

You can specify fields in WITH DATA *datafield-name* that are not indexed. This allows more queries to be satisfied from the index without going to the Master Map. The tradeoff is how many indexes you want to maintain; and that adding data to an index makes it quite a bit larger, which will slow down operations that don't need the data.

You can specify fields in WITH DATA *datafield-name* that are defined in the superclass for the table's persistent class.

The UNIQUE Keyword

Using the UNIQUE keyword, you can specify that each record in the index has a unique value. More specifically, this ensures that no two records within the index (and hence in the table that contains the index) can have the same *collated* value. By default, most indexes use uppercase string collation (to make searches not case-sensitive). In this case, the values "Smith" and "SMITH" are considered to be equal and not unique. **CREATE INDEX** cannot specify non-default index string collation. You can specify a different string collation for individual indexes by defining the index in the class definition.

You can change the namespace default collation to make fields/properties case-sensitive by default. Changing this option requires recompiling all classes and rebuilding all indexes in the namespace. Go to the Management Portal, select the

Classes option, select the namespace for your stored queries and use the **Compile** option to recompile the corresponding classes. Then rebuild all indexes. They will be case-sensitive.

CAUTION: Do not rebuild indexes while the table's data is being accessed by other users. Doing so may result in inaccurate query results.

The BITMAP Keyword

Using the BITMAP keyword, you can specify that this index will be a bitmap index. A bitmap index consists of one or more bit strings in which the bit position represents the row id, and each bit value represents the presence (1) or absence (0) of a specific value for the field in that row (or the value for the combined *field-name* fields). InterSystems SQL maintains these positional bits (as compressed bit strings) when inserting, updating, or deleting data; there is no significant difference in the performance of INSERT, UPDATE, or DELETE operations between using a bitmap index and a regular index. A bitmap index is highly efficient for many types of query operations. They have the following characteristics:

- You can only define bitmap indexes in tables (classes) that either use system-assigned RowID with positive integer values, or use a primary key IDKEY to define custom ID values when the IDKEY is based on a single property with type %Integer and MINVAL > 0, or type %Numeric with SCALE = 0 and MINVAL > 0.

  You can use the **$SYSTEM.SQL.Util.SetOption()** method SET
  status=$SYSTEM.SQL.Util.SetOption("BitmapFriendlyCheck",1,.oldval) to set a system-wide configuration parameter to check at compile time for this restriction, determining whether a defined bitmap index is allowed in a %Storage.SQL class. This check only applies to classes that use %Storage.SQL. The default is 0. You can use **$SYSTEM.SQL.Util.GetOption("BitmapFriendlyCheck")** to determine the current configuration of this option.

  You can only define a bitmap index for tables that use default (%Storage.Persistent) structure. Tables with compound keys, such as a child table, cannot use a bitmap index. If you use DDL (as opposed to using class definitions) to create a table, it meets this requirement and you can make use of bitmap indexes.

- A bitmap index should only be used when the number of possible distinct field values is limited and relatively small. For example, a bitmap index is a good choice for a field for gender, or nationality, or timezone. A bitmap should not be used on a field with the UNIQUE constraint. A bitmap should not be used if a field can have more than 10,000 distinct values, or if multiple indexed fields can have more than 10,000 distinct values.

- Bitmap indexes are very efficient when used in combination with logical AND and OR operations in a WHERE clause. If two or more fields are commonly queried in combination, it may be advantageous to define bitmap indexes for those fields.

For more details, see Bitmap Indices.

The BITMAPEXTENT Keyword

A bitmap extent index is a bitmap index for the table itself. InterSystems SQL uses this index to improve performance of COUNT(*), which returns the number of records (rows) in the table. A table can have, at most, one bitmap extent index. Attempting to create more than one bitmap extent index results in an SQLCODE -400 error with the %msg ERROR #5445: Multiple Extent indexes defined: DDLBEIndex.

All tables defined using CREATE TABLE automatically define a bitmap extent index. This automatically generated index is assigned the Index Name DDLBEIndex and the SQL MapName %%DDLBEIndex. A table defined as a class may have a bitmap extent index defined with an Index Name and SQL MapName of $ClassName.

You can use **CREATE BITMAPEXTENT INDEX** to add a bitmap extent index to a table, or to rename an automatically-generated bitmap extent index. The *index-name* you specify should be the class name corresponding to the *table-name* of the table. This becomes the SQL MapName for the index. No *field-name* or WITH DATA clause can be specified.

The following example creates a bitmap extent index with Index Name DDLBEIndex and the SQL MapName Patient. If Sample.Patient already had a %%DDLBEIndex bitmap extent index, this example renames that index to SQL MapName Patient:

SQL

```
CREATE BITMAPEXTENT INDEX Patient ON TABLE Sample.Patient
```

For more details, see Bitmap Extent Index.

The BITSLICE Keyword

Using the BITSLICE keyword, you can specify that this index will be a bitslice index. A bitslice index is used exclusively for numeric data which is used in calculations. A bitslice index represents each numeric data value as a binary bit string. Rather than indexing a numeric data value using a boolean flag (as in a bitmap index), a bitslice index creates a bit string for each numeric value, a separate bit string for each record. This is a highly specialized type of index that should only be used for fast aggregate calculations. For example, the following would be a candidate for a bitslice index:

SQL

```
SELECT SUM(Salary) FROM Sample.Employee
```

You can create a bitslice index for a string data field, but the bitslice index will represent these data values as canonical numbers. In other words, any non-numeric string, such as "abc" will be indexed as 0. This type of bitslice index could be used to rapidly count records that have a value for a string field and not count those that are NULL.

A bitslice index should not be used in a WHERE clause, because they are not used by the SQL query optimizer.

Populating and maintaining a bitslice index using INSERT, UPDATE, or DELETE operations is significantly slower than using a bitmap index or a regular index. Using several bitslice indexes, and/or using a bitslice index on a field that is frequently updated may have a significant performance cost.

A bitslice index can only be used for records that have system-assigned row Ids with positive integer values. A bitslice index can only be used on a single *field-name*. You cannot specify a WITH DATA clause.

For more details, see Bitslice Indices.

The COLUMNAR Keyword

Using the COLUMNAR keyword, you can specify that this index will be a columnar index. A columnar index is used for a column that is frequently queried but whose table has an underlying row storage structure. By default, each row of a table is stored as a **$LIST** in a separate global subscript. For more details, see Columnar Indexes and Choose an SQL Table Storage Layout.

Rebuilding an Index

Creating an index using the **CREATE INDEX** statement automatically builds the index. However, there are cases when you may wish to explicitly rebuild an index.

CAUTION: You must take additional steps when rebuilding an index if the table's data is being accessed by other users. Failing to do so may result in inaccurate query results. For more details, refer to Building Indexes with BUILD INDEX.

You can build/re-build indexes as follows:

• Using the BUILD INDEX SQL command.

• Using the Management Portal to rebuild all of the indexes for a specified class (table).

• Using the %BuildIndices() method.

To rebuild all indexes for an inactive table, execute the following:

ObjectScript

```
SET status = ##class(myschema.mytable).%BuildIndices()
```

By default, this command purges the indexes prior to rebuilding them. You can override this purge default and use the **%PurgeIndices()** method to explicitly purge specified indexes. If you call **%BuildIndices()** for a range of ID values, InterSystems IRIS does not purge indexes by default.

You can also purge/rebuild specified indexes:

ObjectScript

```
SET status = ##class(myschema.mytable).%BuildIndices($ListBuild("NameIDX","SpouseIDX"))
```

You may want to purge/rebuild an index if the index is corrupt or to change the case sensitivity of the index, as described above. To recompress a bitmap index, use the %SYS.Maint.Bitmap methods, rather than purge/rebuild.

For more details, see Building Indices.

Examples

The following example creates a table named Fred, and then creates an index named "FredIndex" (by stripping out the punctuation from the supplied name "Fred_Index") on the Lastword and Firstword fields of the Fred table.

SQL

```
CREATE TABLE Fred (
  TESTNUM      INT NOT NULL,
  FIRSTWORD   CHAR (30) NOT NULL,
  LASTWORD    CHAR (30) NOT NULL,
  CONSTRAINT FredPK PRIMARY KEY (TESTNUM))

CREATE INDEX Fred_Index
ON TABLE Fred (LASTWORD,FIRSTWORD)
```

The following example creates an index, named "CityIndex" on the City field of the Staff table:

SQL

```
CREATE INDEX CityIndex ON Staff (City)
```

The following example creates an index, named "EmpIndex" on the EmpName field of the Staff table. The UNIQUE constraint is used to avoid having rows with identical values in the fields:

SQL

```
CREATE UNIQUE INDEX EmpIndex ON TABLE Staff (EmpName)
```

The following example creates a bitmap index, named "SKUIndex" on the SKU field of the Purchases table. The BITMAP keyword indicates that this is a bitmap index:

SQL

```
CREATE BITMAP INDEX SKUIndex ON TABLE Purchases (SKU)
```

See Also

- BUILD INDEX command

- DROP INDEX command

- [SEARCH_INDEX](#) function

- [Defining Tables](#)

- [Defining and Building Indexes](#)

- [Using Indexes](#)

- [SQL and Object Settings Pages](#)

- [SQLCODE error messages](#)

- %Library.FunctionalIndex

CREATE METHOD (SQL)

Creates a method in a class.

Synopsis

```
CREATE [STATIC] METHOD name (parameter_list)
    [ characteristics ]
    [ LANGUAGE SQL ]
    BEGIN
code_body ;
    END

CREATE [STATIC] METHOD name (parameter_list)
    [ characteristics ]
    LANGUAGE OBJECTSCRIPT
    { code_body }
```

Description

The **CREATE METHOD** statement creates a class method. This class method may or may not be a stored procedure. To create a method in a class that is exposed as an SQL stored procedure, you must specify the PROCEDURE keyword. By default, **CREATE METHOD** does not create a method which is also a stored procedure; the CREATE PROCEDURE statement always creates a method which is also a stored procedure.

The optional STATIC keyword is provided to clarify that the method created is a static (class) method, not an instance method. This keyword provides no actual functionality.

In order to create a method, you must have %CREATE_METHOD administrative privilege, as specified by the GRANT command. If you are attempting to create a method for an existing class with a defined owner, you must be logged in as the owner of the class. Otherwise, the operation fails with an SQLCODE -99 error.

You cannot create a method in a class if the class definition is a deployed class. This operation fails with an SQLCODE -400 error with the %msg `Unable to execute DDL that modifies a deployed class: 'classname'`.

The following two examples both show the creation of the same class method. The first example uses **CREATE METHOD**, the second defines the class method in the class User.Letters:

SQL

```
CREATE METHOD RandCaseLetter(IN caps CHAR)
  RETURNS INTEGER
  PROCEDURE
LANGUAGE OBJECTSCRIPT
{
:Top
 IF caps="U" {SET x=$RANDOM(91) IF x>64 {QUIT $CHAR(x)}
   ELSE {GOTO Top}}
 ELSEIF caps="L" {SET x=$RANDOM(123) IF x>97 {QUIT $CHAR(x)}
   ELSE {GOTO Top}}
 ELSE {QUIT "case must be 'U' or 'L'"}
}

Class User.Letters Extends %Persistent [ DdlAllowed ]
{
 ClassMethod RandCaseLetter(caps) As %String [ SqlName = RandomLetter, SqlProc ]
  {
    Top
    IF caps="U" {SET x=$RANDOM(91) IF x>64 {QUIT $CHAR(x)}
         ELSE {GOTO Top}}
      ELSEIF caps="L" { SET x=$RANDOM(123)  IF x>97 {QUIT $CHAR(x)}
       ELSE {GOTO Top}}
       ELSE {QUIT "case must be 'U' or 'L'"}
  }
}
```

For information on calling methods from within SQL statements, refer to User-defined Functions. For calling SQL stored procedures in a variety of contexts, refer to the CALL statement.

Arguments

name

The name of the method to be created. This name may be unqualified (StoreName) and take the system-wide default schema name, or qualified by specifying the schema name (Patient.StoreName). You can use the **$SYSTEM.SQL.Schema.Default()** method to determine the current system-wide default schema name. The initial system-wide default schema name is SQLUser which corresponds to the class package name User.

Note that the FOR characteristic (described below) overrides the class name specified in *name*. If a method with this name already exists, the operation fails with an SQLCODE -361 error. To avoid this error, use the optional keyword **OR REPLACE** to modify or replace the existing method. **CREATE OR REPLACE METHOD** has the same effect as invoking **DROP METHOD** to delete the old version of the method and then invoking **CREATE METHOD**.

The name of the generated class is the package name corresponding to the schema name, followed by a dot, followed by "meth", followed by the specified *name*. For example, if the unqualified method name RandomLetter takes the initial default schema SQLUser, the resulting class name would be: User.methRandomLetter. For further details, see SQL to Class Name Transformations.

InterSystems SQL does not allow you to specify a duplicate method name that differs only in letter case. Specifying a method name that differs only in letter case from an existing method name results in an SQLCODE -400 error.

parameter-list

A list of parameters used to pass values to the method. The parameter list is enclosed in parentheses, and parameter declarations in the list are separated by commas. The parentheses are mandatory, even when specifying no parameters. Each parameter declaration in the list consists of (in order):

- An optional keyword specifying whether the parameter mode is IN (input value), OUT (output value), or INOUT (modify value). If omitted, the default parameter mode is IN.

- The parameter name. Parameter names are case-sensitive.

- The data type of the parameter.

- *Optional*: A default value for the parameter. You can specify the DEFAULT keyword followed by a default value; the DEFAULT keyword is optional. If no default is specified, the assumed default is NULL.

The output value from a method is automatically converted from Logical format to Display/ODBC format.

An input value to a method is, by default, not converted from Display/ODBC format to Logical format. However, input display-to-logical conversion can be configured systemwide using the **$SYSTEM.SQL.Util.SetOption("SQLFunctionArgConversion")** method. You can use **$SYSTEM.SQL.Util.GetOption("SQLFunctionArgConversion")** to determine the current configuration of this option.

characteristics

The available keywords are as follows:

FOR *className*	Specifies the name of the class in which to create the method. If the class does not exist, it will be created. You can also specify a class name by qualifying the method name. The class name specified in the FOR clause overrides a class name specified by qualifying the method name.
FINAL	Specifies that subclasses cannot override the method. By default, methods are not final. The FINAL keyword is inherited by subclasses.
PRIVATE	Specifies that the method can only be invoked by other methods of its own class or subclasses. By default, a method is public, and can be invoked without restriction. This restriction is inherited by subclasses.
PROCEDURE	Specifies that the method is an SQL stored procedure. Stored procedures are inherited by subclasses. (This keyword can be abbreviated as PROC.)
RESULT SETS DYNAMIC RESULT SETS [*n*]	Specifies that the method created will contain the ReturnResultsets keyword. All forms of this *characteristics* phrase are synonyms.
RETURNS *datatype*	Specifies the data type of the value returned by a call to the method. If RETURNS is omitted, the method cannot return a value. This specification is inherited by subclasses, and can be modified by subclasses. This *datatype* can specify type parameters such as MINVAL, MAXVAL, and SCALE. For example `RETURNS DECIMAL(19,4)`. Note that when returning a value, InterSystems IRIS ignores the length of *datatype*; for example, `RETURNS VARCHAR(32)` can receive a string of any length that is returned by a call to the method.
SELECTMODE *mode*	Only used when LANGUAGE is SQL (the default). When specified, InterSystems IRIS adds an #SQLCOMPILE SELECT=*mode* statement to the corresponding class method, thus generating the SQL statements defined in the method with the specified SELECTMODE. The possible *mode* values are LOGICAL, ODBC, RUNTIME, and DISPLAY. The default is LOGICAL.

If you specify a query keyword (such as CONTAINSID or RESULTS) that is not valid for a method, the system generates an SQLCODE -47 error. If you specify a duplicate query keyword (such as FINAL FINAL), the system generates an SQLCODE -44 error.

The SELECTMODE clause is used for **SELECT** query operations and for **INSERT** and **UPDATE** operations. It specifies the compile-time select mode. The value that you specify for SELECTMODE is added at the beginning of the ObjectScript class method code as: #sqlcompile select=*mode*. For further details, see #sqlcompile select.

- In a **SELECT** query, the SELECTMODE specifies the mode in which data is returned. If the *mode* value is LOGICAL, then logical (internal storage) values are returned. For example, dates are returned in $HOROLOG format. If the *mode* value is ODBC, logical-to-ODBC conversion is applied, and ODBC format values are returned. If the *mode* value is DISPLAY, logical-to-display conversion is applied, and display format values are returned. If the *mode* value is RUNTIME, the display mode can be set (to LOGICAL, ODBC, or DISPLAY) at execution time.

- In an INSERT or UPDATE operation, the SELECTMODE RUNTIME option supports automatic conversion of input data values from a display format (DISPLAY or ODBC) to logical storage format. This compiled display-to-logical data conversion code is applied only if the select mode setting when the SQL code is executed is LOGICAL (which is the default for all InterSystems SQL execution interfaces).

When the SQL code is executed, the %SQL.Statement class *%SelectMode* property specifies the execution-time select mode, as described in Using Dynamic SQL. For further details on SelectMode options, refer to Data Display Options.

LANGUAGE

A keyword clause specifying the language you are using for *code_body*. Permitted clauses are LANGUAGE OBJECTSCRIPT (for ObjectScript) or LANGUAGE SQL. If the LANGUAGE clause is omitted, SQL is the default.

code_body

The program code for the method to be created. You specify this code in either SQL or ObjectScript. The language used must match the LANGUAGE clause. However, code specified in ObjectScript can contain embedded SQL.

InterSystems IRIS uses the code you supply to generate the actual code of the method.

If the code you specify is SQL, InterSystems IRIS provides additional lines of code when generating the method that embed the SQL in an ObjectScript "wrapper," provide a procedure context handler (if necessary), and handle return values. The following is an example of this InterSystems IRIS-generated wrapper code:

ObjectScript

```
NEW SQLCODE,%ROWID,%ROWCOUNT,title
&sql( SELECT col FROM tbl )
QUIT $GET(title)
```

If the code you specify is OBJECTSCRIPT, the ObjectScript code must be enclosed in curly braces. All code lines must be indented from column 1, except for labels and macro preprocessor directives. A label or macro directive must be prefaced by a colon (:) in column 1.

For ObjectScript code, you must explicitly define the "wrapper" (which NEWs variable and uses QUIT exit and (optionally) to return a value upon completion).

The method can be exposed as a stored procedure by specifying the PROCEDURE keyword. When a stored procedure is called, an object of the class %Library.SQLProcContext is instantiated in the %sqlcontext variable. This procedure context handler is used to pass the procedure context back and forth between the procedure and its caller (for example, the ODBC server).

%sqlcontext consists of several properties, including an Error object, the SQLCODE error status, the SQL row count, and an error message. The following example shows the values used to set several of these:

```
SET %sqlcontext.%SQLCODE=SQLCODE
SET %sqlcontext.%ROWCOUNT=%ROWCOUNT
SET %sqlcontext.%Message=%msg
```

The values of SQLCODE and %ROWCOUNT are automatically set by the execution of an SQL statement. The %sqlcontext object is reset before each execution.

Alternatively, an error context can be established by instantiating a %SYSTEM.Error object and setting it as %sqlcontext.Error.

Examples

The following two examples both show the creation of the same class method. The first example uses **CREATE METHOD**, the second defines the class method in the class User.Letters:

SQL

```
CREATE METHOD RandCaseLetter(IN caps CHAR)
  RETURNS INTEGER
  PROCEDURE
LANGUAGE OBJECTSCRIPT
{
:Top
 IF caps="U" {SET x=$RANDOM(91) IF x>64 {QUIT $CHAR(x)}
   ELSE {GOTO Top}}
 ELSEIF caps="L" {SET x=$RANDOM(123) IF x>97 {QUIT $CHAR(x)}
   ELSE {GOTO Top}}
 ELSE {QUIT "case must be 'U' or 'L'"}
}

Class User.Letters Extends %Persistent [ DdlAllowed ]
{
 ClassMethod RandCaseLetter(caps) As %String [ SqlName = RandomLetter, SqlProc ]
  {
   Top
   IF caps="U" {SET x=$RANDOM(91) IF x>64 {QUIT $CHAR(x)}
       ELSE {GOTO Top}}
     ELSEIF caps="L" { SET x=$RANDOM(123)  IF x>97 {QUIT $CHAR(x)}
      ELSE {GOTO Top}}
      ELSE {QUIT "case must be 'U' or 'L'"}
  }
}
```

The following example specifies two input parameters, both of which have default values. The optional DEFAULT keyword is specified for the first parameter, omitted for the second parameter:

SQL

```
CREATE METHOD RandomLetter(IN firstlet CHAR DEFAULT 'A',IN lastlet CHAR 'Z')
BEGIN
-- SQL program code
END
```

The following example uses **CREATE METHOD** with SQL code to generate the method UpdateSalary in the class Sample.Employee:

The following example uses **CREATE METHOD** with SQL code to generate the method UpdateSalary in the class Sample.Employee:

SQL

```
CREATE METHOD UpdateSalary ( IN SSN VARCHAR(11), IN Salary INTEGER )
   FOR Sample.Employee
   BEGIN
     UPDATE Sample.Employee SET Salary = :Salary WHERE SSN = :SSN;
   END
```

The following example creates the RandomLetter() method stored as a procedure that generates a random capital letter. You can then invoke this method as a function in a **SELECT** statement. A **DROP METHOD** is provided to delete the RandomLetter() method.

SQL

```
CREATE METHOD RandomLetter()
RETURNS INTEGER
PROCEDURE
LANGUAGE OBJECTSCRIPT
{
:Top
 SET x=$RANDOM(91)
 IF x<65 {GOTO Top}
 ELSE {QUIT $CHAR(x)}
}
```

SQL

```
SELECT Name FROM Sample.Person
WHERE Name %STARTSWITH RandomLetter()
```

SQL

```
DROP METHOD RandomLetter
```

The following Embedded SQL example uses **CREATE METHOD** with ObjectScript code to generate the method TraineeTitle in the class SQLUser.MyStudents and returns a *Title* value:

ObjectScript

```
 &sql(CREATE METHOD TraineeTitle(
  IN SSN VARCHAR(11),
  INOUT Title VARCHAR(50) )
   RETURNS VARCHAR(30)
   FOR SQLUser.MyStudents
   LANGUAGE OBJECTSCRIPT
   {
       NEW SQLCODE,%ROWCOUNT
       &sql(SELECT Title INTO :Title FROM Sample.Employee
           WHERE SSN = :SSN)
       IF $GET(%sqlcontext)'= "" {
          SET %sqlcontext.%SQLCODE=SQLCODE
          SET %sqlcontext.%ROWCOUNT=%ROWCOUNT }
          QUIT
   })
   IF SQLCODE=0 { WRITE !,"Created a method" QUIT}
   ELSEIF SQLCODE=-361 { WRITE !,"Method already exists SQLCODE: ",SQLCODE
     &sql(DROP METHOD TraineeTitle FROM SQLUser.MyStudents)
     IF SQLCODE=0 { WRITE !,"Dropped a method" QUIT}}
   ELSE { WRITE !,"SQL error: ",SQLCODE }
```

It uses the %sqlcontext object, and sets its %SQLCODE and %ROWCOUNT properties using the corresponding SQL variables. Note the curly braces enclosing the ObjectScript code following the method's LANGUAGE OBJECTSCRIPT keyword. Within the ObjectScript code there is Embedded SQL code, marked by &sql and enclosed in parentheses.

Security and Privileges

The CREATE METHOD command is a privileged operation that requires the user to have %Development:USE permission. Such permissions can be granted through the Management Portal. Executing a CREATE METHOD command without these privileges will result in an SQLCODE -99 error and the command will fail.

Users without proper permissions can still execute this command under one of two conditions:

- The command is executed via Embedded SQL, which does not perform privilege checks.

- The user explicitly specifies no privilege checking by, for example, calling either **%Prepare()** with the checkPriv argument set to 0 or **%ExecDirectNoPriv()** on a %SQL.Statement.

See Also

- CALL

- CREATE PROCEDURE

- DROP METHOD

- Defining and Using Stored Procedures

CREATE ML CONFIGURATION (SQL)

Creates an ML configuration.

Synopsis

```
CREATE [ OR REPLACE ] ML CONFIGURATION ml-configuration-name PROVIDER provider-name
        [ %DESCRIPTION description] [ USING json-object-string ]
        [ provider-connection-settings ]
```

Arguments

ml-configuration-name	The name for the ML configuration being created. A valid identifier, subject to the same additional naming restrictions as a table name. An ML configuration name is unqualified (*mlconfig-name*). An unqualified ML configuration name takes the default schema name.
PROVIDER *provider-name*	A string specifying the name of a machine learning provider, where values are: • AutoML • H2O • DataRobot • PMML
%DESCRIPTION *description*	*Optional* — String. A text description for the ML configuration. See details below.
USING *json-object-string*	*Optional* — A JSON string specifying one or more key-value pairs; see details below.
provider-connection-settings	Any additional settings, required for connection, that vary by the machine learning provider. See details below.

Description

The **CREATE ML CONFIGURATION** command creates an ML configuration for training models. You can specify one or more of the following properties:

- The provider (required)

- The description

- The USING clause

- Provider connection settings

ML Configuration Description

%DESCRIPTION accepts a text string enclosed in single quotes, which you can use to provide a description for documenting your configuration. This text can be of any length, and can contain any characters, including blank spaces.

USING

You can specify a default **USING** clause for your configuration. This clause accepts a JSON string with one or more key-value pairs. When **TRAIN MODEL** is executed, by default the **USING** clause of the configuration is used.

You must make sure that the parameters you specify are recognized by the provider you select. Failing to do so may result in an error when training.

An example with H2O as the provider:

```
CREATE ML CONFIGURATION h2o_config PROVIDER H2O USING {"seed":100, "nfolds":4}
```

Provider Connection Settings

Depending on the provider specified by your configuration, there may be additional fields you must enter to establish a successful connection.

DataRobot

You must specify the following values to successfully connect to DataRobot:

- URL [=] *url-string* — where *url-string* is the URL of a DataRobot endpoint.

- APITOKEN [=] *token-string* — where *token-string* is your client API token to access the DataRobot AutoML server.

A complete ML configuration for DataRobot could be created with a query as follows:

```
CREATE ML CONFIGURATION datarobot-configuration PROVIDER DataRobot1 URL url-string APITOKEN token-string
```

With proper values for *url-string* and *token-string*

Required Security Privileges

Calling **CREATE ML CONFIGURATION** requires %CREATE_ML_CONFIGURATION privileges; otherwise, there is a SQLCODE –99 error (Privilege Violation). To assign %CREATE_ML_CONFIGURATION privileges, use the GRANT command.

Configuration Naming Conventions

Configuration names follow identifier conventions, subject to the restrictions below. By default, configuration names are simple identifiers. A configuration name should not exceed 256 characters. Configuration names are not case-sensitive. For further details, see Identifiers.

InterSystems IRIS® uses the configuration name to generate a corresponding class name. A class name contains only alphanumeric characters (letters and numbers) and must be unique within the first 96 characters. To generate this class name, InterSystems IRIS first strips punctuation characters from the configuration name, and then generates an identifier that is unique within the first 96 characters, substituting an integer (beginning with 0) for the final character when needed to create a unique class name. InterSystems IRIS generates a unique class name from a valid configuration name, but this name generation imposes the following restrictions on the naming of configurations:

- A configuration name must include at least one letter. Either the first character of the view name or the first character after initial punctuation characters must be a letter

- InterSystems IRIS supports 16-bit (wide) characters for configuration names. A character is a valid letter if it passes the *$ZNAME* test.

- If the first character of the configuration name is a punctuation character, the second character cannot be a number. This results in an SQLCODE -400 error, with a %msg value of "ERROR #5053: Class name 'schema.name' is invalid" (without the punctuation character). For example, specifying the configuration name %7A generates the %msg "ERROR #5053: Class name 'User.7A' is invalid".

- Because generated class names do not include punctuation characters, it is not advisable (though possible) to create a configuration name that differs from an existing configuration name only in its punctuation characters. In this case,

InterSystems IRIS substitutes an integer (beginning with 0) for the final character of the name to create a unique class name.

- A configuration name may be much longer than 96 characters, but configuration names that differ in their first 96 alphanumeric characters are much easier to work with.

A configuration name can only be unqualified. An unqualified configuration name (`viewname`) takes the system-wide default schema name.

If you would like to redefine an ML configuration to use the same name, you can specify the OR REPLACE option to replace a pre-existing ML configuration with different behavior.

Examples

```
CREATE ML CONFIGURATION autoML_config PROVIDER AutoML %DESCRIPTION 'my AutoML configuration!'
```

See Also

- ALTER ML CONFIGURATION, DROP ML CONFIGURATION

CREATE MODEL (SQL)

Creates a model definition.

Synopsis

```
CREATE MODEL [ IF NOT EXISTS ] model-name PREDICTING ( label-column )
     FROM model-source [ USING json-object ]

CREATE MODEL [ IF NOT EXISTS ] model-name PREDICTING ( label-column )
     WITH feature-column-clause [ USING json-object ]

CREATE MODEL [ IF NOT EXISTS ] model-name PREDICTING ( label-column )
     WITH feature-column-clause FROM model-source [ USING json-object ]
```

Arguments

This synopsis shows the valid forms of CREATE MODEL. The CREATE MODEL command must have either a FROM or WITH clause (or both).

model-name	The name for the model definition being created. A valid identifier, subject to the same additional naming restrictions as a table name. A model name is unqualified (`modelname`). An unqualified model name takes the default schema name.
PREDICTING (*label-column*)	The name of the column being predicted, aka, the label column. A standard identifier. See details below.
WITH *feature-column-clause*	Inputs to the model, aka the feature columns, as either the name of a column and it's datatype or as a comma-separated list of the names of columns and datatypes. Each column name is a standard identifier.
FROM *model-source*	The table or view from which the model is being built. This can be a table, view, or results of a join.
USING *json-object-string*	*Optional* — A JSON string specifying one or more key-value pairs. See more details below.

Description

The **CREATE MODEL** command creates a model definition of the structure specified. This includes, at a minimum:

- The model name
- The label column
- The feature column(s)

Predicting

You must specify the output column (or label column) that your model predicts, given the input columns (or feature columns). For example, if you are designing a `SpamFilter` model which identifies emails that are spam mail, you may have a label column named `IsSpam`, which is a boolean value designating whether a given email is spam or not. You can also specify the data type of this column; otherwise, IntegratedML infers the type:

```
CREATE MODEL SpamFilter PREDICTING (IsSpam) FROM EmailData
CREATE MODEL SpamFilter PREDICTING (IsSpam binary) FROM EmailData
```

WITH and FROM

A model definition must contain a **WITH** and/or **FROM** to specify the schema characteristics of the model.

WITH

Using **WITH**, you can specify which input columns (features) to include in your model definition. Note that you must specify the data type of each column, even when using a **FROM** clause in your statement:

```
CREATE MODEL SpamFilter PREDICTING (IsSpam) WITH (email_length int, subject_title varchar)
CREATE MODEL SpamFilter PREDICTING (IsSpam) WITH (email_length int, subject_title varchar) FROM EmailData
```

FROM

FROM allows you to use every single column from a specified table or view, without having to identify each column individually:

```
CREATE MODEL SpamFilter PREDICTING (IsSpam) FROM EmailData
```

This clause is fully general, and can specify any subquery expression. IntegratedML infers the data types of each column. By using **FROM**, you supply a default data set for future **TRAIN MODEL** statements using this model definition. You can use **FROM** along with **WITH** to both supply a default data set and to explicitly name feature columns.

Without a **WITH** clause, IntegratedML infers the data types of each column, and implicitly uses the result of the **FROM** clause as if it were the following query:

```
SELECT * FROM model-source
```

USING

You can specify a default **USING** clause for your model definition. This clause accepts a JSON string with one or more key-value pairs. When **TRAIN MODEL** is executed, by default the **USING** clause of the model definition is used. All parameters specified in the **USING** clause of your ML configuration overwrite those same parameters in the **USING** clause of your model definition.

You must make sure that the parameters you specify are recognized by the provider you select. Failing to do so may result in an error when training.

Required Security Privileges

Calling **CREATE MODEL** requires %MANAGE_MODEL privileges; otherwise, there is a SQLCODE –99 error (Privilege Violation). To assign %MANAGE_MODEL privileges, use the GRANT command.

Model Naming Conventions

Model names follow identifier conventions, subject to the restrictions below. By default, model names are simple identifiers. A model name should not exceed 256 characters. Model names are not case-sensitive.

InterSystems IRIS uses the model name to generate a corresponding class name. A class name contains only alphanumeric characters (letters and numbers) and must be unique within the first 96 characters. To generate this class name, InterSystems IRIS first strips punctuation characters from the model name, and then generates an identifier that is unique within the first 96 characters, substituting an integer (beginning with 0) for the final character when needed to create a unique class name. InterSystems IRIS generates a unique class name from a valid model name, but this name generation imposes the following restrictions on the naming of models:

- A model name must include at least one letter. Either the first character of the view name or the first character after initial punctuation characters must be a letter

- InterSystems IRIS supports 16-bit (wide) characters for model names. A character is a valid letter if it passes the *$ZNAME* test.

- If the first character of the model name is a punctuation character, the second character cannot be a number. This results in an SQLCODE -400 error, with a %msg value of "ERROR #5053: Class name 'schema.name' is invalid" (without the punctuation character). For example, specifying the model name %7A generates the %msg "ERROR #5053: Class name 'User.7A' is invalid".

- Because generated class names do not include punctuation characters, it is not advisable (though possible) to create a model name that differs from an existing model name only in its punctuation characters. In this case, InterSystems IRIS substitutes an integer (beginning with 0) for the final character of the name to create a unique class name.

- A model name may be much longer than 96 characters, but model names that differ in their first 96 alphanumeric characters are much easier to work with.

A model name can only be unqualified. An unqualified model name (`viewname`) takes the system-wide default schema name.

Examples

```
CREATE MODEL PatientReadmit PREDICTING (IsReadmitted) FROM patient_table USING {"seed": 3}
CREATE MODEL PatientReadmit PREDICTING (IsReadmitted) WITH (age, gender, encounter_type, admit_reason,
 starttime, endtime, prior_visits, diagnosis, comorbitities)
```

See Also

- ALTER MODEL, DROP MODEL, TRAIN MODEL

CREATE PROCEDURE (SQL)

Creates a method or query which is exposed as an SQL stored procedure.

Synopsis

```
CREATE PROCEDURE procname(parameter_list)
    [ characteristics ]
    [ LANGUAGE SQL ]
    BEGIN code_body ;
    END
CREATE PROCEDURE procname(parameter_list)
    [ characteristics ]
    LANGUAGE OBJECTSCRIPT
    { code_body }
CREATE PROCEDURE procname(parameter_list)
    [ characteristics ]
    LANGUAGE { JAVA | PYTHON | DOTNET } EXTERNAL NAME external-stored-procedure
```

Description

The **CREATE PROCEDURE** statement creates a method or a query which is, by default, exposed as an SQL stored procedure. A stored procedure can be invoked by all processes in the current namespace. Stored procedures are inherited by subclasses.

- If LANGUAGE SQL, the *code_body* must contain a **SELECT** statement in order to generate a query exposed as a stored procedure. If the code does not contain a **SELECT** statement, **CREATE PROCEDURE** creates a method.

- If LANGUAGE OBJECTSCRIPT, the *code_body* must call **Execute()** and **Fetch()** methods in order to generate a query exposed as a stored procedure. It may also call **Close()**, **FetchRows()**, and **GetInfo()** methods. If the code does not call **Execute()** and **Fetch()**, **CREATE PROCEDURE** creates a method.

To create a method not exposed as a stored procedure, use the CREATE METHOD or CREATE FUNCTION statement. To create a query not exposed as a stored procedure, use the CREATE QUERY statement. These statements can also be used to create a method or query exposed as a stored procedure by specifying the PROCEDURE characteristic keyword.

In order to create a procedure, you must have %CREATE_PROCEDURE administrative privilege, as specified by the GRANT command. If you are attempting to create a procedure for an existing class with a defined owner, you must be logged in as the owner of the class. Otherwise, the operation fails with an SQLCODE -99 error.

You cannot create a procedure in a class if the class definition is a deployed class. This operation fails with an SQLCODE -400 error with the %msg Unable to execute DDL that modifies a deployed class: 'classname'.

A stored procedure is executed using the CALL statement.

For information on calling methods from within SQL statements, refer to User-defined Functions.

Arguments

procname

The name of the method or query to be created as a stored procedure. The *procname* must be followed by parentheses, even if no parameters are specified. A procedure name can take any of the following forms:

- Unqualified: Takes the default schema name. For example, MedianAgeProc().

- Qualified: Supplies a schema name. For example, Patient.MedianAgeProc().

- Multilevel: Qualified with one or more schema levels to parallel corresponding class package members. In this case, the *procname* may contain only one period character; the other periods in the corresponding class method name are replaced with underline characters. The period is specified before the lowest level class package member. For example, %SYSTEM.SQL_GetROWID(), or %SYS_PTools.StatsSQL_Export().

An unqualified *procname* takes the default schema name. You can use the **$SYSTEM.SQL.Schema.Default()** method to determine the current system-wide default schema name. The initial system-wide default schema name is SQLUser which corresponds to the class package name User.

Note that the FOR characteristic (described below) overrides the class name specified in *procname*. If a procedure with this name already exists, the operation fails with an SQLCODE -361 error.

InterSystems SQL uses the SQL *procname* to generate a corresponding class name. This name consists of the package name corresponding to the schema name, followed by a dot, followed by "proc", followed by the specified procedure name. For example, if the unqualified procedure name RandomLetter() takes the default schema SQLUser, the resulting class name would be: User.procRandomLetter(). For further details, see SQL to Class Name Transformations.

InterSystems SQL does not allow you to specify a *procname* that differs only in letter case. Specifying a *procname* that differs only in letter case from an existing procedure name results in an SQLCODE -400 error.

If the specified *procname* already exists in the current namespace, the system generates an SQLCODE -361 error. To determine if a specified *procname* already exists in the current namespace, use the **$SYSTEM.SQL.Schema.ProcedureExists()** method.

Include the optional keyword **OR REPLACE** to modify or replace an existing procedure without generating an error. **CREATE OR REPLACE PROCEDURE** has the same effect as invoking **DROP PROCEDURE** to delete the old version of the procedure and then invoking **CREATE PROCEDURE**.

Note: InterSystems SQL procedure names and InterSystems TSQL procedure names share the same set of names. Therefore, you cannot create an SQL procedure that has the same name as a TSQL procedure in the same namespace. Attempting to do so results in an SQLCODE -400 error.

parameter_list

A list of parameters used to pass values to the method or query. The parameter list is enclosed in parentheses, and parameter declarations in the list are separated by commas. The parentheses are mandatory, even if you specify no parameters.

Each parameter declaration in the list consists of (in order):

- An optional keyword specifying whether the parameter mode is IN (input value), OUT (output value), or INOUT (modify value). If omitted, the default parameter mode is IN.

- The parameter name. Parameter names are case-sensitive.

- The data type of the parameter.

- *Optional*: A default value for the parameter. You can specify the DEFAULT keyword followed by a default value; the DEFAULT keyword is optional. If no default is specified, the assumed default is NULL.

The following example creates a stored procedure with two input parameters, both of which have default values. One input parameter specifies the optional DEFAULT keyword, the other input parameter omits this keyword:

SQL

```
CREATE PROCEDURE AgeQuerySP(IN topnum INT DEFAULT 10,IN minage INT 20)
   BEGIN
   SELECT TOP :topnum Name,Age FROM Sample.Person
   WHERE Age > :minage ;
   END
```

The following example is functionally identical to the example above. The optional DEFAULT keyword is omitted:

SQL

```
CREATE PROCEDURE AgeQuerySP(IN topnum INT 10,IN minage INT 20)
   BEGIN
   SELECT TOP :topnum Name,Age FROM Sample.Person
   WHERE Age > :minage ;
   END
```

The following are all valid **CALL** statements for this procedure: CALL AgeQuerySP(6,65); CALL AgeQuerySP(6); CALL AgeQuerySP(,65); CALL AgeQuerySP().

The following example creates a method exposed as a stored procedure with three parameters:

SQL

```
CREATE PROCEDURE UpdatePaySP
  (IN Salary INTEGER DEFAULT 0,
   IN Name VARCHAR(50),
   INOUT PayBracket VARCHAR(50) DEFAULT 'NULL')
BEGIN
   UPDATE Sample.Person SET Salary = :Salary
   WHERE Name=:Name ;
END
```

A stored procedure does not perform automatic format conversion of parameters. For example, an input parameter in ODBC format or Display format remains in that format. It is the responsibility of the code that calls the procedure, and the procedure code itself, to handle IN/OUT values in a format appropriate to the application, and to perform any necessary conversions.

Because the method or query is exposed as a stored procedure, it uses a procedure context handler to pass the procedure context back and forth between the procedure and its caller. When a stored procedure is called, an object of the class %Library.SQLProcContext is instantiated in the %sqlcontext variable. This is used to pass the procedure context back and forth between the procedure and its caller (for example, the ODBC server).

%sqlcontext consists of several properties, including an Error object, the SQLCODE error status, the SQL row count, and an error message. The following example shows the values used to set several of these:

```
SET %sqlcontext.%SQLCODE=SQLCODE
SET %sqlcontext.%ROWCOUNT=%ROWCOUNT
SET %sqlcontext.%Message=%msg
```

The values of SQLCODE and %ROWCOUNT are automatically set by the execution of an SQL statement. The %sqlcontext object is reset before each execution.

Alternatively, an error context can be established by instantiating a %SYSTEM.Error object and setting it as %sqlcontext.Error.

characteristics

Different *characteristics* are used for creating a method than those used to create a query.

If you specify a *characteristics* that is not valid, the system generates an SQLCODE -47 error. Specifying duplicate *characteristics* results in an SQLCODE -44 error.

The available method *characteristics* keywords are as follows:

Method Keyword	Meaning
FOR *className*	Specifies the name of the class in which to create the method. If the class does not exist, it will be created. You can also specify a class name by qualifying the method name. The class name specified in the FOR clause overrides a class name specified by qualifying the method name.
	If you specify the class name using the FOR my.class syntax, InterSystems IRIS defines the class method with Sqlname=procname. Therefore, the method should be invoked as **my.procname()** (*not* **my.class_procname()**).

Method Keyword	Meaning
FINAL	Specifies that subclasses cannot override the method. By default, methods are not final. The FINAL keyword is inherited by subclasses.
PRIVATE	Specifies that the method can only be invoked by other methods of its own class or subclasses. By default, a method is public, and can be invoked without restriction. This restriction is inherited by subclasses.
RESULT SETS DYNAMIC RESULT SETS [*n*]	Specifies that the method created will contain the ReturnResultsets keyword. All forms of this *characteristics* phrase are synonyms.
RETURNS *datatype*	Specifies the data type of the value returned by a call to the method. If RETURNS is omitted, the method cannot return a value. This specification is inherited by subclasses, and can be modified by subclasses. This *datatype* can specify type parameters such as MINVAL, MAXVAL, and SCALE. For example `RETURNS DECIMAL(19,4)`. Note that when returning a value, InterSystems IRIS ignores the length of *datatype*; for example, `RETURNS VARCHAR(32)` can receive a string of any length that is returned by a call to the method.
SELECTMODE *mode*	Only used when LANGUAGE is SQL (the default). When specified, InterSystems IRIS adds an #SQLCOMPILE SELECT=*mode* statement to the corresponding class method, thus generating the SQL statements defined in the method with the specified SELECTMODE. The possible *mode* values are LOGICAL, ODBC, RUNTIME, and DISPLAY. The default is LOGICAL.

The available query *characteristics* keywords are as follows:

Query Keyword	Description
CONTAINID *integer*	Specifies which field, if any, returns the ID. Set CONTAINID to the number of the column that returns the ID, or 0 if no column returns the ID. InterSystems IRIS does not validate that the named field actually contains the ID, so a user error here results in inconsistent data.
FOR *className*	Specifies the name of the class in which to create the method. If the class does not exist, it will be created. You can also specify a class name by qualifying the method name. The class name specified in the FOR clause overrides a class name specified by qualifying the method name.
FINAL	Specifies that subclasses cannot override the method. By default, methods are not final. The FINAL keyword is inherited by subclasses.

Query Keyword	Description
RESULTS (*result_set*)	Specifies the data fields in the order that they are returned by the query. If you specify a RESULTS clause, you must list all fields returned by the query as a comma-separated list enclosed in parentheses. Specifying fewer or more fields than are returned by the query results in a SQLCODE -76 cardinality mismatch error.
	For each field you specify a column name (which will be used as the column header) and a data type.
	If LANGUAGE SQL, you can omit the RESULTS clause. If you omit the RESULTS clause, the ROWSPEC is automatically generated during class compilation.
SELECTMODE *mode*	Specifies the mode used to compile the query. The possible values are LOGICAL, ODBC, RUNTIME, and DISPLAY. The default is RUNTIME.

The SELECTMODE clause is used for **SELECT** query operations and for **INSERT** and **UPDATE** operations. It specifies the compile-time select mode. The value that you specify for SELECTMODE is added at the beginning of the ObjectScript class method code as: #sqlcompile select=*mode*. For further details, see #sqlcompile select.

- In a **SELECT** query, the SELECTMODE specifies the mode in which data is returned. If the *mode* value is LOGICAL, then logical (internal storage) values are returned. For example, dates are returned in $HOROLOG format. If the *mode* value is ODBC, logical-to-ODBC conversion is applied, and ODBC format values are returned. If the *mode* value is DISPLAY, logical-to-display conversion is applied, and display format values are returned. If the *mode* value is RUNTIME, the display mode can be set (to LOGICAL, ODBC, or DISPLAY) at execution time.

- In an INSERT or UPDATE operation, the SELECTMODE RUNTIME option supports automatic conversion of input data values from a display format (DISPLAY or ODBC) to logical storage format. This compiled display-to-logical data conversion code is applied only if the select mode setting when the SQL code is executed is LOGICAL (which is the default for all InterSystems SQL execution interfaces).

When the SQL code is executed, the %SQL.Statement class *%SelectMode* property specifies the execution-time select mode, as described in Using Dynamic SQL. For further details on SelectMode options, refer to Data Display Options.

The RESULTS clause specifies the results of a query. The SQL data type parameters in the RESULTS clause are translated into corresponding InterSystems IRIS data type parameters in the query's ROWSPEC. For example, the RESULTS clause RESULTS (Code VARCHAR(15)) generates a ROWSPEC specification of ROWSPEC = "Code:%Library.String(MAXLEN=15)".

LANGUAGE

A keyword clause specifying the procedure code language. Available options are:

- LANGUAGE OBJECTSCRIPT (for ObjectScript) or LANGUAGE SQL. The procedure code is specified in the *code_body*.

- LANGUAGE JAVA, LANGUAGE PYTHON, or LANGUAGE DOTNET for an SQL procedure that invokes an external stored procedure in one of these languages. The syntax for an external stored procedure is as follows:

```
LANGUAGE langname EXTERNAL NAME external-routine-name
```

Where *langname* is JAVA, PYTHON, or DOTNET and *external-routine-name* is a quoted string containing the name an external routine in the specified language. The SQL procedure invokes an existing routine; you cannot write code in these languages within the **CREATE PROCEDURE** statement. Stored procedure libraries in these languages are

stored external to IRIS, and therefore do not have to be packaged, imported, or compiled within IRIS. The following is an example of a **CREATE PROCEDURE** invoking an existing JAVA external stored procedure:

```
CREATE PROCEDURE updatePrice (item_name VARCHAR, new_price INTEGER)
LANGUAGE JAVA
EXTERNAL NAME 'Orders.updatePrice'
```

If the LANGUAGE clause is omitted, SQL is the default.

code_body

The program code for the method or query to be created. You specify this code in either SQL or ObjectScript. The language used must match the LANGUAGE clause. However, code specified in ObjectScript can contain embedded SQL. InterSystems IRIS uses the code you supply to generate the actual code of the method or query.

- SQL program code is prefaced with a BEGIN keyword, followed by the SQL code itself. At the end of each complete SQL statement, specify a semicolon (;). A query contains only one SQL statement—a **SELECT** statement. You can also create procedures that insert, update, or delete data. SQL program code concludes with an END keyword.

  Input parameters are specified in the SQL statement as host variables, with the form `:name`. (Note that you *should not* use question marks (?) to specify input parameters in the SQL code. The procedure will successfully build, but when it is called these parameters cannot be passed or take default values.)

- ObjectScript program code is enclosed within curly braces: `{ code }`. Lines of code must be indented. If specified, a label or a `#include` preprocessor command must be prefaced by a colon and appear in the first column, as shown in the following example:

  SQL

```
CREATE PROCEDURE SP123()
  LANGUAGE OBJECTSCRIPT
{
:Top
:#include %occConstant
  WRITE "Hello World"
  IF 0=$RANDOM(2) { GOTO Top }
  ELSE {QUIT $$$OK }
}
```

  The system automatically includes %occInclude. If program code contains InterSystems IRIS Macro Preprocessor statements (# commands, ## functions, or $$$macro references) the processing and expansion of these statements is part of the procedure's method definition, and get processed and expanded when the method is compiled. For more details on preprocessor commands, see Preprocessor Directives Reference.

InterSystems IRIS provides additional lines of code when generating the procedure that embed the SQL in an ObjectScript "wrapper," provide a procedure context handler, and handle return values. The following is an example of this InterSystems IRIS-generated wrapper code:

ObjectScript

```
NEW SQLCODE,%ROWID,%ROWCOUNT,title
&sql(
     -- code_body
    )
QUIT $GET(title)
```

If the code you specify is OBJECTSCRIPT, you must explicitly define the "wrapper" (which NEWs variable and uses **QUIT val** to return a value upon completion.

Examples

The examples that follow are divided into those that use an SQL *code_body*, and those that use an ObjectScript *code_body*.

Examples Using SQL Code

The following example creates a simple query, named PersonStateSP, exposed as a stored procedure. It declares no parameters and takes default values for *characteristics* and LANGUAGE:

ObjectScript

```
WRITE !,"Creating a procedure"
&sql(CREATE PROCEDURE PersonStateSP() BEGIN
     SELECT Name,Home_State FROM Sample.Person ;
     END)
IF SQLCODE=0 { WRITE !,"Created a procedure" }
ELSEIF SQLCODE=-361 { WRITE !,"Procedure already exists" }
ELSE { WRITE !,"SQL error: ",SQLCODE }
```

You can go to the Management Portal, select the **Classes** option, then select the SAMPLES namespace. There you will find the stored procedure created by the above example: User.procPersonStateSP.cls. From this display you can delete this procedure before rerunning the above program example. You can, of course, use **DROP PROCEDURE** to delete a procedure:

ObjectScript

```
WRITE !,"Deleting a procedure"
&sql(DROP PROCEDURE SAMPLES.PersonStateSP)
IF SQLCODE=0 { WRITE !,"Deleted a procedure" }
ELSEIF SQLCODE=-362 { WRITE !,"Procedure did not exist" }
ELSE { WRITE !,"SQL error: ",SQLCODE }
```

The following example creates a procedure to update data. It uses **CREATE PROCEDURE** to generate the method UpdateSalary in the class Sample.Employee:

SQL

```
CREATE PROCEDURE UpdateSalary ( IN SSN VARCHAR(11), IN Salary INTEGER )
   FOR Sample.Employee
   BEGIN
     UPDATE Sample.Employee SET Salary = :Salary WHERE SSN = :SSN;
   END
```

Examples Using ObjectScript Code

The following example creates the RandomLetterSP() stored procedure method that generates a random capital letter. You can then invoke this method as a function in a **SELECT** statement. A **DROP PROCEDURE** is provided to delete the RandomLetterSP() method.

SQL

```
CREATE PROCEDURE RandomLetterSP()
RETURNS INTEGER
LANGUAGE OBJECTSCRIPT
{
:Top
 SET x=$RANDOM(90)
 IF x<65 {GOTO Top}
 ELSE {QUIT $CHAR(x)}
}
```

SQL

```
SELECT Name FROM Sample.Person
WHERE Name %STARTSWITH RandomLetterSP()
```

SQL

```
DROP PROCEDURE RandomLetterSP
```

The following **CREATE PROCEDURE** example uses ObjectScript calls to the **Execute()**, **Fetch()**. and **Close()** methods. Such procedures may also contain **FetchRows()** and **GetInfo()** method calls:

SQL

```
CREATE PROCEDURE GetTitle()
    FOR Sample.Employee
    RESULTS (ID %Integer)
    CONTAINID 1
    LANGUAGE OBJECTSCRIPT
    Execute(INOUT qHandle %Binary)
    {  QUIT 1 }
    Fetch(INOUT qHandle %Binary, INOUT Row %List, INOUT AtEnd %Integer)
    {  QUIT 1 }
    Close(INOUT qHandle %Binary)
    {  QUIT 1 }
```

The following **CREATE PROCEDURE** example uses an ObjectScript call to the %SQL.Statement result set class:

SQL

```
CREATE PROCEDURE Sample_Employee.GetTitle(
    INOUT Title VARCHAR(50) )
    RETURNS VARCHAR(30)
    FOR Sample.Employee
    LANGUAGE OBJECTSCRIPT
  {
  SET myquery="SELECT TOP 10 Name,Title FROM Sample.Employee"
  SET tStatement = ##class(%SQL.Statement).%New()
  SET qStatus = tStatement.%Prepare(myquery)
    IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET rset = tStatement.%Execute()
  DO rset.%Display()
  WRITE !,"End of data"
  }
```

If the ObjectScript code block fetches data into a local variable (for example, *Row*), you must conclude the code block with the line `SET Row=""` to indicate an end-of-data condition.

The following example uses **CREATE PROCEDURE** with ObjectScript code that includes Embedded SQL. It generates the method GetTitle in the class Sample.Employee and passes out the *Title* value as a parameter:

SQL

```
CREATE PROCEDURE Sample_Employee.GetTitle(
   IN SSN VARCHAR(11),
   INOUT Title VARCHAR(50) )
    RETURNS VARCHAR(30)
    FOR Sample.Employee
    LANGUAGE OBJECTSCRIPT
    {
        NEW SQLCODE,%ROWCOUNT
        &sql(SELECT Title INTO :Title FROM Sample.Employee
            WHERE SSN = :SSN)
        IF $GET(%sqlcontext)'= "" {
           SET %sqlcontext.%SQLCODE=SQLCODE
           SET %sqlcontext.%ROWCOUNT=%ROWCOUNT }
           QUIT
     }
```

It uses the %sqlcontext object, and sets its %SQLCODE and %ROWCOUNT properties using the corresponding SQL variables. Note the curly braces enclosing the ObjectScript code following the procedure's LANGUAGE OBJECTSCRIPT keyword. Within the ObjectScript code there is Embedded SQL code, marked by `&sql` and enclosed in parentheses.

Security and Privileges

The CREATE PROCEDURE command is a privileged operation that requires the user to have %Development:USE permission. Such permissions can be granted through the Management Portal. Executing a CREATE PROCEDURE command without these privileges will result in an SQLCODE -99 error and the command will fail.

Users without proper permissions can still execute this command under one of two conditions:

- The command is executed via Embedded SQL, which does not perform privilege checks.

- The user explicitly specifies no privilege checking by, for example, calling either **%Prepare()** with the checkPriv argument set to 0 or **%ExecDirectNoPriv()** on a %SQL.Statement.

See Also

- SELECT

- CALL

- DROP PROCEDURE

- CREATE METHOD, CREATE FUNCTION

- GRANT

- Defining and Using Stored Procedures

- Querying the Database

CREATE QUERY (SQL)

Creates a query.

Synopsis

```
CREATE [OR REPLACE] QUERY queryname(parameter_list)
     [ characteristics ] [ LANGUAGE SQL ]
      BEGIN code_body ;
     END
CREATE QUERY queryname(parameter_list) [characteristics]
       LANGUAGE OBJECTSCRIPT
    { code_body }
```

Description

The **CREATE QUERY** statement creates a query in a class. By default, a query named MySelect would be stored as User.queryMySelect or SQLUser.queryMySelect.

CREATE QUERY creates a query which may or may not be exposed as a stored procedure. To create a query that is exposed as a stored procedure, you must specify the PROCEDURE keyword as one of its *characteristics*. You can also use the CREATE PROCEDURE statement to create a query which is exposed as a stored procedure.

In order to create a query, you must have %CREATE_QUERY administrative privilege, as specified by the GRANT command. If you are attempting to create a query for an existing class with a defined owner, you must be logged in as the owner of the class. Otherwise, the operation fails with an SQLCODE -99 error.

You cannot create a query in a class if the class definition is a deployed class. This operation fails with an SQLCODE -400 error with the %msg Unable to execute DDL that modifies a deployed class: 'classname'.

Arguments

queryname

The name of the query to be created in a stored procedure class. The *queryname* must be a valid identifier and must be followed by parentheses, even if no parameters are specified. The procedure name may be unqualified (StoreName) and take the default schema name, or qualified by specifying the schema name (Patient.StoreName). You can use the **$SYSTEM.SQL.Schema.Default()** method to determine the current system-wide default schema name. The initial system-wide default schema name is SQLUser which corresponds to the class package name User.

Note that the FOR characteristic (described below) overrides the class name specified in *queryname*. If a method with this name already exists, the operation fails with an SQLCODE -361 error.

The name of the generated class is the package name corresponding to the schema name, followed by a dot, followed by "query", followed by the specified *queryname*. For example, if the unqualified query name RandomLetter takes the initial default schema SQLUser, the resulting class name would be: User.queryRandomLetter. For further details, see SQL to Class Name Transformations.

InterSystems SQL does not allow you to specify a *queryname* that differs only in letter case. Specifying a *queryname* that differs only in letter case from an existing query name results in an SQLCODE -400 error.

If the specified *queryname* already exists in the current namespace, the system generates an SQLCODE -361 error.

Include the optional keyword **OR REPLACE** to modify or replace an existing query without generating an error. **CREATE OR REPLACE QUERY** has the same effect as invoking **DROP QUERY** to delete the old version of the query and then invoking **CREATE QUERY**.

parameter-list

A list of parameter declarations for parameters used to pass values to the query. The parameter list is enclosed in parentheses, and parameter declarations in the list are separated by commas. The parentheses are mandatory, even if you specify no parameters.

Each parameter declaration in the list consists of (in order):

- An optional keyword specifying whether the parameter mode is IN (input value), OUT (output value), or INOUT (modify value). If omitted, the default parameter mode is IN.

- The parameter name. Parameter names are case-sensitive.

- The data type of the parameter.

- *Optional*: A default value for the parameter. You can specify the DEFAULT keyword followed by a default value; the DEFAULT keyword is optional. If no default is specified, the assumed default is NULL.

The following example creates a query exposed as a stored procedure with two input parameters, both of which have default values. The *topnum* input parameter specifies the optional DEFAULT keyword; the *minage* input parameter omits this keyword:

SQL

```
CREATE QUERY AgeQuery(IN topnum INT DEFAULT 10,IN minage INT 20)
    PROCEDURE
    BEGIN
    SELECT TOP :topnum Name,Age FROM Sample.Person
    WHERE Age > :minage ;
    END
```

The following are all valid **CALL** statements for this query: `CALL AgeQuery(6,65);` `CALL AgeQuery(6);` `CALL AgeQuery(,65);` `CALL AgeQuery().`

characteristics

An optional argument denoting one or more keywords that specify the characteristics of a query. Characteristics can be specified in any order. The available *characteristics* keywords are as follows:

Characteristics Keyword	Description
CONTAINID *integer*	Specifies which field, if any, returns the ID. Set CONTAINID to the number of the column that returns the ID, or 0 if no column returns the ID. InterSystems IRIS does not validate that the named field actually contains the ID, so a user error here results in inconsistent data.
FOR *className*	Specifies the name of the class in which to create the method. If the class does not exist, it will be created. You can also specify a class name by qualifying the method name. The class name specified in the FOR clause overrides a class name specified by qualifying the method name.
FINAL	Specifies that subclasses cannot override the method. By default, methods are not final. The FINAL keyword is inherited by subclasses.
PROCEDURE	Specifies that the query is an SQL stored procedure. Stored procedures are inherited by subclasses. (This keyword can be abbreviated as PROC.)

Characteristics Keyword	Description
RESULTS (*result_set*)	Specifies the data fields in the order that they are returned by the query. If you specify a RESULTS clause, you must list all fields returned by the query as a comma-separated list enclosed in parentheses. Specifying fewer or more fields than are returned by the query results in a SQLCODE -76 cardinality mismatch error.
	For each field you specify a column name (which will be used as the column header) and a data type.
	If LANGUAGE SQL, you can omit the RESULTS clause. If you omit the RESULTS clause, the ROWSPEC is automatically generated during class compilation.
SELECTMODE *mode*	Specifies the mode used to compile the query. The possible values are LOGICAL, ODBC, RUNTIME, and DISPLAY. The default is RUNTIME.

If you specify a method keyword (such as PRIVATE or RETURNS) that is not valid for a query, the system generates an SQLCODE -47 error. Specifying duplicate *characteristics* results in an SQLCODE -44 error.

The SELECTMODE clause specifies the mode in which data is returned. If the *mode* value is LOGICAL, then logical (internal storage) values are returned. For example, dates are returned in $HOROLOG format. If the *mode* value is ODBC, logical-to-ODBC conversion is applied, and ODBC format values are returned. If the *mode* value is DISPLAY, logical-to-display conversion is applied, and display format values are returned. If the *mode* value is RUNTIME, the mode can be set (to LOGICAL, ODBC, or DISPLAY) at execution time by setting the %SQL.Statement class *%SelectMode* property, as described in Using Dynamic SQL. The RUNTIME mode default is LOGICAL. For further details on SelectMode options, refer to Data Display Options. The value that you specify for SELECTMODE is added at the beginning of the ObjectScript class method code as: #SQLCompile SELECT=*mode*. For further details, see #sqlcompile select.

The RESULTS clause specifies the results of a query. The SQL data type parameters in the RESULTS clause are translated into corresponding InterSystems IRIS data type parameters in the query's ROWSPEC. For example, the RESULTS clause RESULTS (Code VARCHAR(15)) generates a ROWSPEC specification of ROWSPEC = "Code:%Library.String(MAXLEN=15)".

LANGUAGE

An optional keyword clause specifying the language you are using for *code_body*. Permitted clauses are LANGUAGE OBJECTSCRIPT or LANGUAGE SQL. If the LANGUAGE clause is omitted, SQL is the default.

If the LANGUAGE is SQL a class query of type %Library.SQLQuery is generated. If the LANGUAGE is OBJECTSCRIPT, a class query of type %Library.Query is generated.

code_body

The program code for the query to be created. You specify this code in either SQL or ObjectScript. The language used must match the LANGUAGE clause. However, code specified in ObjectScript can contain embedded SQL.

If the code you specify is SQL, it must consist of a single **SELECT** statement. The program code for a query in SQL is prefaced with a BEGIN keyword, followed by the program code (a **SELECT** statement). At the end of the program code, specify a semicolon (;) then an END keyword.

If the code you specify is OBJECTSCRIPT, it must contain calls to the **Execute()** and **Fetch()** class methods of the %Library.Query class provided by InterSystems IRIS, and may contain **Close()**, **FetchRows()**, and **GetInfo()** method calls. ObjectScript code is enclosed in curly braces. If **Execute()** or **Fetch()** are missing, an SQLCODE -46 error is generated upon compilation.

If the ObjectScript code block fetches data into a local variable (for example, *Row*), you must conclude the code block with the line SET Row="" to indicate an end-of-data condition.

If the query is exposed as a stored procedure (by specifying the PROCEDURE keyword in *characteristics*), it uses a procedure context handler to pass the procedure context back and forth between the procedure and its caller.

When a stored procedure is called, an object of the class %Library.SQLProcContext is instantiated in the %sqlcontext variable. This is used to pass the procedure context back and forth between the procedure and its caller (for example, the ODBC server).

%sqlcontext consists of several properties, including an Error object, the SQLCODE error status, the SQL row count, and an error message. The following example shows the values used to set several of these:

```
SET %sqlcontext.%SQLCODE=SQLCODE
SET %sqlcontext.%ROWCOUNT=%ROWCOUNT
SET %sqlcontext.%Message=%msg
```

The values of SQLCODE and %ROWCOUNT are automatically set by the execution of an SQL statement. The %sqlcontext object is reset before each execution.

Alternatively, an error context can be established by instantiating a %SYSTEM.Error object and setting it as %sqlcontext.Error.

InterSystems IRIS uses the code you supply to generate the actual code of the query.

Examples

The following example creates a query named DocTestPersonState. It declares no parameters, sets the SELECTMODE *characteristic*, and takes the default (SQL) for LANGUAGE:

SQL

```
CREATE QUERY DocTestPersonState() SELECTMODE RUNTIME
BEGIN
SELECT Name,Home_State FROM Sample.Person ;
END
```

You can go to the Management Portal, select the **Classes** option, then select the SAMPLES namespace. There you will find the query created by the above example: User.queryDocTestPersonState.cls. From this display you can delete this query before rerunning the above program example. You can, of course, use **DROP QUERY** to delete created queries.

The following Embedded SQL example creates a method-based query named DocTestSQLCODEList which fetches a list of SQLCODEs and their descriptions. It sets a RESULTS result set characteristic, sets LANGUAGE as ObjectScript, and calls the **Execute()**, **Fetch()**, and **Close()** methods:

ObjectScript

```
&sql(CREATE QUERY DocTestSQLCODEList()
RESULTS (SQLCODE SMALLINT,Description VARCHAR(100))
  PROCEDURE
  LANGUAGE OBJECTSCRIPT
Execute(INOUT QHandle BINARY(255))
  {
   SET QHandle=1,%i(QHandle)=""
   QUIT ##lit($$$OK)
  }
Fetch(INOUT QHandle BINARY(255), INOUT Row %List, INOUT AtEnd INT)
  {
   SET AtEnd=0,Row=""
   SET %i(QHandle)=$o(^%qCacheSQL("SQLCODE",%i(QHandle)))
   IF %i(QHandle)="" {SET AtEnd=1 QUIT ##lit($$$OK) }
     SET Row=$lb(%i(QHandle),^%qCacheSQL("SQLCODE",%i(QHandle),1,1))
   QUIT ##lit($$$OK)
  }
Close(INOUT QHandle BINARY(255))
  {
   KILL %i(QHandle)
   QUIT ##lit($$$OK)
  }
)
IF SQLCODE=0 { WRITE !,"Created a query" }
ELSEIF SQLCODE=-361 { WRITE !,"Query exists: ",%msg }
ELSE { WRITE !,"CREATE QUERY error: ",SQLCODE }
```

You can go to the Management Portal, select the **Classes** option, then select the SAMPLES namespace. There you will find the query created by the above example: User.queryDocTestSQLCODEList.cls. From this display you can delete this query before rerunning the above program example. You can, of course, use **DROP QUERY** to delete created queries.

The following Dynamic SQL example creates a query named DocTest, then executes this query using the **%PrepareClassQuery()** method of the %SQL.Statement class:

ObjectScript

```
/* Creating the Query */
set myquery=4
set myquery(1)="CREATE QUERY DocTest() SELECTMODE RUNTIME "
set myquery(2)="BEGIN "
set myquery(3)="SELECT TOP 5 Name,Home_State FROM Sample.Person ; "
set myquery(4)="END"
set tStatement = ##class(%SQL.Statement).%New()

set qStatus = tStatement.%Prepare(.myquery)
if $$$ISERR(qStatus) {write "%Prepare failed:" do $SYSTEM.Status.DisplayError(qStatus) quit}

set rset = tStatement.%Execute()
if (rset.%SQLCODE '= 0) {write "%Unable to call query", !, "SQLCODE ", rset.%SQLCODE, ": ",
rset.%Message quit}

/* Calling the Query */
write !,"Calling a class query",!
set cqStatus = tStatement.%PrepareClassQuery("User.queryDocTest","DocTest")
if $$$ISERR(cqStatus) {write "%PrepareClassQuery failed:" do $SYSTEM.Status.DisplayError(cqStatus)
quit}

set rset = tStatement.%Execute()
if (rset.%SQLCODE '= 0) {write "Unable to call class query", !, "SQLCODE ", rset.%SQLCODE, ": ",
rset.%Message quit}

write "Query data",!,!
while rset.%Next()
{
    do rset.%Print()
}
if (rset.%SQLCODE < 0) {write "%Next failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message
quit}
write !,"End of data"

/* Deleting the Query */
&sql(DROP QUERY DocTest)
if SQLCODE = 0 {write !,"Deleted the query"}
```

For further details, refer to Dynamic SQL.

Security and Privileges

The CREATE QUERY command is a privileged operation that requires the user to have %Development:USE permission. Such permissions can be granted through the Management Portal. Executing a CREATE QUERY command without these privileges will result in an SQLCODE -99 error and the command will fail.

Users without proper permissions can still execute this command under one of two conditions:

- The command is executed via Embedded SQL, which does not perform privilege checks.

- The user explicitly specifies no privilege checking by, for example, calling either **%Prepare()** with the checkPriv argument set to 0 or **%ExecDirectNoPriv()** on a %SQL.Statement.

See Also

- SELECT

- CALL

- DROP QUERY

- CREATE PROCEDURE

- [Querying the Database](#)
- [Defining and Using Stored Procedures](#)

CREATE ROLE (SQL)

Creates a role.

Synopsis

```
CREATE ROLE role-name
```

Description

The **CREATE ROLE** command creates a role. A role is a named set of privileges that may be assigned to multiple users. A role may be assigned to multiple users, and a user may be assigned multiple roles. A role is available system-wide, it is not limited to a specific namespace.

A *role-name* can be any valid identifier of up to 64 characters. A *role-name* must follow identifier naming conventions. A role name can contain Unicode characters. Role names are not case-sensitive. A *role-name* can be a delimited identifier enclosed in quotation marks, if the **Support Delimited Identifiers** configuration option is checked (the default). If a delimited identifier, *role-name* can be an SQL reserved word. It can contain a period (.), caret (^), and the two-character arrow sequence (->). It cannot contain a comma (,) or a colon (:) character. It may begin with any valid character, except the asterisk (*).

When initially created, a role is just a name; it has no privileges. To add privileges to a role, use the GRANT command. You can also use the **GRANT** command to assign one or more roles to a role. This permits you to create a hierarchy of roles.

If you invoke **CREATE ROLE** to create a role that already exists, SQL issues an SQLCODE -118 error. You can determine if a role already exists by invoking the **$SYSTEM.SQL.Security.RoleExists()** method:

ObjectScript

```
WRITE $SYSTEM.SQL.Security.RoleExists("%All"),!
WRITE $SYSTEM.SQL.Security.RoleExists("Madmen")
```

This method returns 1 if the specified role exists, and 0 if the role does not exist. Role names are not case-sensitive.

To delete a role, use the **DROP ROLE** command.

Privileges

The **CREATE ROLE** command is a privileged operation. Before using **CREATE ROLE** in embedded SQL, you must be logged in as a user with one of the following:

- The %Admin_Secure administrative resource with USE permission.

- The %Admin_RoleEdit administrative resource with USE permission.

- Full security privileges on the system.

If you are not, the **CREATE ROLE** command results in an SQLCODE -99 error (Privilege Violation). Use the **$SYSTEM.Security.Login()** method to assign a user with appropriate privileges:

ObjectScript

```
DO $SYSTEM.Security.Login(username,password)
&sql(      )
```

You must have the **%Service_Login:Use** privilege to invoke the **$SYSTEM.Security.Login()** method. For further information, refer to %SYSTEM.Security in the *InterSystems Class Reference*.

Arguments

role-name

The name of the role to be created, which is an identifier. Role names are not case-sensitive.

Examples

The following examples attempt to create a role named BkUser. The user "FRED" in the first example does not have create role privileges. The user "_SYSTEM" in the second example does have create role privileges.

ObjectScript

```
  DO $SYSTEM.Security.Login("FRED","FredsPassword")
  &sql(CREATE ROLE BkUser)
  IF SQLCODE=-99 {
    WRITE !,"You don't have CREATE ROLE privileges" }
  ELSEIF SQLCODE=-118 {
    WRITE !,"The role already exists" }
  ELSE {
    WRITE !,"Created a role. Error code is: ",SQLCODE }
```

ObjectScript

```
  DO $SYSTEM.Security.Login("_SYSTEM","SYS")
Main
  &sql(CREATE ROLE BkUser)
  IF SQLCODE=-99 {
    WRITE !,"You don't have CREATE ROLE privileges" }
  ELSEIF SQLCODE=-118 {
    WRITE !,"The role already exists" }
  ELSE {
    WRITE !,"Created a role. Error code is: ",SQLCODE }
Cleanup
  SET toggle=$RANDOM(2)
  IF toggle=0 {
    &sql(DROP ROLE BkUser)
    WRITE !,"DROP USER error code: ",SQLCODE
  }
  ELSE {
    WRITE !,"No drop this time"
    QUIT
  }
```

(The **$RANDOM** toggle is provided so that you can execute this example program repeatedly.)

See Also

- SQL statements: DROP ROLE, CREATE USER, DROP USER, GRANT, REVOKE, %CHECKPRIV

- SQL Users, Roles, and Privileges

- SQLCODE error messages

- ObjectScript: $ROLES and $USERNAME special variables

CREATE SCHEMA (SQL)

Creates a schema.

Synopsis

```
CREATE SCHEMA [ IF NOT EXISTS ] name
```

Arguments

Argument	Description
name	The name of the schema being created. The name is an identifier.
IF NOT EXISTS	*Optional* — Suppresses the error that arises if a schema with *name* already exists. The schema is not re-created.

Description

Creates a schema definition, along with a corresponding package definition. The owner of the schema will be defined as the user who issues this command. A schema created in this manner will not appear in INFORMA-TION_SCHEMA.SCHEMATA until a table has been created within the schema.

If IF NOT EXISTS was specified and the schema already exists, this command performs no action. If IF NOT EXISTS was not specified but a schema with the same name already exists, SQLCODE -476 is returned.

See Also

- DROP SCHEMA

- SQLCODE error messages

CREATE TABLE (SQL)

Creates a table definition.

Synopsis

Basic Table Creation

```
CREATE TABLE [IF NOT EXISTS] table (column type, column2 type2, ...)
CREATE TABLE [IF NOT EXISTS] table AS SELECT query ...
```

Column Constraints

```
CREATE TABLE table (column type NOT NULL, ...)
CREATE TABLE table (column type UNIQUE, ...)
CREATE TABLE table (UNIQUE (column, column2, ...), ...)
CREATE TABLE table (..., CONSTRAINT uniqueName UNIQUE (column, column2, ...))
CREATE TABLE table (column type PRIMARY KEY, ...)
CREATE TABLE table (..., PRIMARY KEY (column, column2, ...))
CREATE TABLE table (..., CONSTRAINT pKeyName PRIMARY KEY (column, column2, ...))
CREATE TABLE table (..., CONSTRAINT fKeyName FOREIGN KEY (column) REFERENCES refTable (refColumn))
CREATE TABLE table (..., CONSTRAINT fKeyName FOREIGN KEY (column, column2, ...) REFERENCES refTable
(refColumn, refColumn2, ...))
CREATE TABLE table (..., CONSTRAINT fKeyName FOREIGN KEY (...) REFERENCES refTable))
CREATE TABLE table (..., CONSTRAINT fKeyName FOREIGN KEY (...) REFERENCES ... ON UPDATE refAction))

CREATE TABLE table (..., CONSTRAINT fKeyName FOREIGN KEY (...) REFERENCES ... ON DELETE refAction))

CREATE TABLE table (..., CONSTRAINT fKeyName FOREIGN KEY (...) REFERENCES ... NOCHECK))
```

Special Columns and Column Properties

```
CREATE TABLE table (column type DEFAULT defaultSpec, ...)
CREATE TABLE table (column type COMPUTECODE [OBJECTSCRIPT | PYTHON ] {code}, ...)
CREATE TABLE table (column type COMPUTECODE ... {code} COMPUTEONCHANGE (column, column2, ...), ...)
CREATE TABLE table (column type COMPUTECODE ... {code} CALCULATED, ...)
CREATE TABLE table (column type COMPUTECODE ... {code} TRANSIENT, ...)
CREATE TABLE table (column type ON UPDATE updateSpec, ...)
CREATE TABLE table (column type IDENTITY, ...)
```

Table Options

```
CREATE TABLE table ...   SHARD
CREATE TABLE table ... SHARD KEY (shardKeyColumn, shardKeyColumn2, ...)
CREATE TABLE table ... SHARD KEY (coshardKeyColumn) COSHARD WITH (coshardTable)
CREATE GLOBAL TEMPORARY TABLE table ...
CREATE TABLE table ... WITH %CLASSPARAMETER pName = pValue, %CLASSPARAMETER pName2 = pValue2, ...
```

Description

The **CREATE TABLE** command creates a table definition of the structure specified. **CREATE TABLE** creates both an SQL table and the corresponding InterSystems IRIS® data platform class. For more details, see Class Definitions of Created Tables.

Note: These syntaxes do not include keywords that are parsed for compatibility only but perform no operation. For more details on these keywords, see Options Supported for Compatibility Only.

Basic Table Creation

You can create a table by specifying column definitions and their data types. Alternatively, you can use a **CREATE TABLE AS SELECT** query to copy column definitions and data from an existing table.

- **CREATE TABLE [IF NOT EXISTS]** *table* (*column type*, *column2 type2*, ...) creates a table containing one or more columns, each of the specified data type.

This statement creates a table with two columns. The first column accepts string values of up to 30 characters. The second column accepts valid dates.

SQL

```
CREATE TABLE Sample.Person (
    Name VARCHAR(30),
    DateOfBirth TIMESTAMP)
```

Example: Create and Populate Table

- **CREATE TABLE [IF NOT EXISTS]** *table* **AS SELECT** *query* copies column definitions and column data from an existing table (or tables) into a new table based on the specified **SELECT** query. The **SELECT** query can specify any combination of tables or views. You may also specify a STORAGETYPE, %CLASSPARAMETER, or sharded table by supplying the relevant clauses.

  This statement creates a new table, `Sample.YoungPeople`, based on a subset of data from the `Sample.People` table with a columnar storage type.

```
CREATE TABLE Sample.YoungPeople
AS SELECT Name,Age
FROM Sample.People
WHERE Age < 21
WITH STORAGETYPE = COLUMNAR
```

When creating a table, the user has the option to include the IF NOT EXISTS condition. Doing so suppresses the error if *table* already exists. For further details, see the following section on methods to check for existing tables.

Column Constraints

Column constraints govern what values are permitted for a column, what the default value is for a column, and whether the column values must be unique. You can also define primary and foreign key constraints on columns. You can specify multiple column constraints per column, in any order. Separate column constraints by a space.

NOT NULL Constraint

- **CREATE TABLE** *table* (*column type* **NOT NULL, ...**) requires all records of the specified column to have a value defined, that is, not be NULL values.

  This statement creates a table where neither column can be null.

  SQL

```
CREATE TABLE Sample.Person (
    Name VARCHAR(30) NOT NULL,
    DateOfBirth TIMESTAMP NOT NULL)
```

The empty string (") is not considered a null value. You can input an empty string into a column that accepts character strings, even if that column is defined with a NOT NULL restriction.

The NULL data constraint keyword (without NOT) explicitly specifies that this column can accept a null value. This is the default definition for a column.

Default Constraint

- **CREATE TABLE** *table* (*column type* **DEFAULT** *defaultSpec***, ...**) specifies the default data value that InterSystems IRIS provides automatically for this column during an **INSERT** operation if the **INSERT** does not supply a data value. If the **INSERT** operation inserts a NULL value into a column that specifies both a DEFAULT value and a NOT NULL constraint, the column uses the DEFAULT value. If the column does not define a NOT NULL constraint, then it uses the NULL value instead of the DEFAULT value.

  This statement sets default values for the `MembershipStatus` and `MembershipTerm` columns.

SQL

```
CREATE TABLE Sample.Member (
    MemberId INT NOT NULL,
    MembershipStatus CHAR(13) NOT NULL DEFAULT 'M',
    MembershipTerm INT NOT NULL DEFAULT 2)
```

Unique Constraints

Unique constraints require that a column can contain only unique values. To see which columns have the unique constraint set, see Catalog Details for a Table.

- **CREATE TABLE** *table* (*column type* **UNIQUE, ...**) constrains the specified column to accept only unique values. No two records can contain the same value for this column.

  This statement sets the unique constraint on the `UserName` column:

  ### SQL

  ```
  CREATE TABLE Sample.People (
      UserName VARCHAR(30) UNIQUE NOT NULL,
      FirstName VARCHAR(30),
      LastName VARCHAR(30))
  ```

  The SQL empty string (") is considered to be a data value, so with the UNIQUE data constraint applied, no two records can contain an empty string value for this column. NULL is not considered to be a data value, so the UNIQUE data constraint does not apply to multiple NULLs. To restrict use of NULL for a column, use the NOT NULL keyword constraint.

  Note: In sharded tables, the unique constraint adds a significant performance cost to inserts and updates. If insert or update performance is important, avoid this constraint or include a shard key for the table. Note that sharded tables have additional restrictions on the UNIQUE constraint.

  For more details on query performance, see Evaluate Unique Constraints and Querying the Sharded Cluster.

- **CREATE TABLE** *table* (**UNIQUE** (*column*, *column2*, ...), ...) requires that all values for a specified group of columns, when concatenated together, result in a unique value. The individual columns do not need to be unique. You can specify this constraint at any location within the comma-separated list of columns being defined.

  This statement requires that the combination of `FirstName` and `LastName` records in the created table are unique, even though `FirstName` and `LastName` records can individually contain duplicates.

  ### SQL

  ```
  CREATE TABLE Sample.People (
      FirstName VARCHAR(30),
      LastName VARCHAR(30),
      UNIQUE (FirstName,LastName))
  ```

- **CREATE TABLE** *table* (**...**, **CONSTRAINT** *uniqueName* **UNIQUE** (*column,column2*, ...)) specifies a name for the UNIQUE constraint. If you want to drop a UNIQUE constraint from a table definition, then the **ALTER TABLE** command requires this constraint name.

  This statement is functionally equivalent to the previous statement and names the constraint `FirstLast`.

  ### SQL

  ```
  CREATE TABLE Sample.People (
      FirstName VARCHAR(30),
      LastName VARCHAR(30),
      CONSTRAINT FirstLast UNIQUE (FirstName,LastName))
  ```

Primary Key Constraints

The PRIMARY KEY constraint designates a column, or combination of columns, as the primary key, constraining that column or columns to be unique and not null. Defining a primary key is optional. When you define a table, InterSystems IRIS automatically creates a generated column, the RowID Column (default name "ID"), which functions as a unique row identifier. For more details on the primary key, see Defining the Primary Key.

- **CREATE TABLE** *table* (*column type* **PRIMARY KEY, ...**) designates a single column in the table as the primary key, constraining it be unique and not null.

  This statement creates a table that designates the EmpNum column as the primary key:

  SQL

  ```
  CREATE TABLE Sample.Employee (
      EmpNum INT PRIMARY KEY,
      NameLast CHAR (30) NOT NULL,
      NameFirst CHAR (30) NOT NULL,
      StartDate TIMESTAMP,
      Salary MONEY)
  ```

  In the **Catalog Details** section of the Management Portal, the generated primary key name has the form *table*PKey*N*, where table is the name of the table and *N* is the constraint count integer.

- **CREATE TABLE** *table* (**..., PRIMARY KEY** (*column*, *column2*, **...**)) designates one or more columns as the primary key. You can specify the PRIMARY KEY clause at any location within the comma-separated list of columns. Specifying a single column in this clause is functionally equivalent to specifying this clause on a specific column by using the previous syntax. If you specify a comma-separated list of columns in this clause, then each column is defined as not null but may contain duplicate values, so long as the combination of the column values is a unique value.

  This statement designates the combination of the FirstName and LastName columns as the primary key:

  SQL

  ```
  CREATE TABLE Sample.People (
      FirstName VARCHAR(30),
      LastName VARCHAR(30),
      PRIMARY KEY (FirstName,LastName))
  ```

- **CREATE TABLE** *table* (**..., CONSTRAINT** *pKeyName* **PRIMARY KEY** (*column*, *column2*, **...**)) enables you to explicitly name your primary key. You can view the name of the primary key from the **Catalog Details** section of the Management Portal.

  This statement is functionally equivalent to the first PRIMARY KEY syntax and additionally names the primary key EmployeePK.

  SQL

  ```
  CREATE TABLE Sample.Employee (
      EmpNum INT,
      NameLast CHAR (30) NOT NULL,
      NameFirst CHAR (30) NOT NULL,
      StartDate TIMESTAMP,
      Salary MONEY,
      CONSTRAINT EmployeePK PRIMARY KEY (EmpNum))
  ```

Foreign Key Constraints

The FOREIGN KEY constraint designates a column, or combination of columns, as a reference to another table. The value stored in the foreign key column uniquely identifies a record in the other table. You can designate more than one foreign key per table. Each foreign key reference must exist in the referenced table and must be defined as unique. The referenced column cannot contain duplicate values or NULL. For more details on foreign keys, see Defining a Foreign Key.

- **CREATE TABLE** *table* **(..., CONSTRAINT** *fKeyName* **FOREIGN KEY (***column***) REFERENCES** *refTable* **(***refColumn***))** designates a column from the table being created as a foreign key that references the *refColumn* column of the *refTable* reference table. The foreign key column and referenced column can have different names but they must have the same data type and column constraints. *fKeyName* specifies the name of the foreign key and is required.

  This statement creates an `Orders` table that defines a foreign key named `CustomersFK`. With this foreign key, the values of the `CustomerNum` column are IDs specified in the `CustID` column of the `Customers` table.

  SQL

  ```
  CREATE TABLE Orders (
      OrderID INT,
      OrderItem VARCHAR,
      OrderQuantity INT,
      CustomerNum INT,
      CONSTRAINT OrdersPK PRIMARY KEY (OrderID),
      CONSTRAINT CustomersFK FOREIGN KEY (CustomerNum) REFERENCES Customers (CustID))
  ```

- **CREATE TABLE** *table* **(..., CONSTRAINT** *fKeyName* **FOREIGN KEY (***column***,** *column2***, ...) REFERENCES** *refTable* **(***refColumn***,** *refColumn2***, ...))** designates a combination of columns as the foreign key of the referenced columns. The foreign key columns and referenced columns must correspond in number of columns and in order listed.

  This statement designates the `CustomerNum` and `SalesPersonNum` column combination of the `Orders` as the foreign key. These column values reference the corresponding `CustID` and `SalespID` columns of the `Customers` table.

  SQL

  ```
  CREATE TABLE Orders (
      OrderID INT,
      OrderItem VARCHAR,
      OrderQuantity INT,
      CustomerNum INT,
      SalesPersonNum INT,
      CONSTRAINT OrdersPK PRIMARY KEY (OrderID),
      CONSTRAINT CustomersFK FOREIGN KEY (CustomerNum,SalesPersonNum) REFERENCES Customers
  (CustID,SalespID))
  ```

- **CREATE TABLE** *table* **(..., CONSTRAINT** *fKeyName* **FOREIGN KEY (...) REFERENCES** *refTable***))** omits the reference column name. The foreign key of the column, or combination of columns, defaults to the primary key of the reference table (if defined), otherwise the IDENTITY column of the reference table (if defined), and otherwise the `RowID` column of the reference table.

  This statement sets a foreign key in which the `CustomerNum` column references the primary key of the `Customers` table, assuming that this table has the primary key defined.

  SQL

  ```
  CREATE TABLE Orders (
      OrderID INT,
      OrderItem VARCHAR,
      OrderQuantity INT,
      CustomerNum INT,
      SalesPersonNum INT,
      CONSTRAINT OrdersPK PRIMARY KEY (OrderID),
      CONSTRAINT CustomersFK FOREIGN KEY (CustomerNum) REFERENCES Customers)
  ```

- **CREATE TABLE** *table* **(..., CONSTRAINT** *fKeyName* **FOREIGN KEY (...) REFERENCES ... ON UPDATE** *refAction***))** defines the UPDATE rule for the reference table. When you attempt to change the primary key value of a row from the reference table, the ON UPDATE clause defines what action to take for the rows in that table. Valid reference action values are NO ACTION (default), SET DEFAULT, SET NULL, and CASCADE. You can specify this clause in conjunction with the ON DELETE clause.

  This statement creates a table that, when the reference column `CustID` is updated, the foreign key column `CustomerNum` receives the same update.

SQL

```
CREATE TABLE Orders (
    OrderID INT,
    OrderItem VARCHAR,
    OrderQuantity INT,
    CustomerNum INT,
    CONSTRAINT OrdersPK PRIMARY KEY (OrderID),
    CONSTRAINT CustomersFK FOREIGN KEY (CustomerNum) REFERENCES Customers (CustID)
        ON UPDATE CASCADE)
```

- **CREATE TABLE** *table* **(..., CONSTRAINT** *fKeyName* **FOREIGN KEY (...) REFERENCES ... ON DELETE** *refAction*)) defines the DELETE rule for the reference table. When you attempt to delete a row from the reference table, the ON DELETE clause defines what action to take for rows in that table. Valid reference action values are NO ACTION (default), SET DEFAULT, SET NULL, and CASCADE. You can specify this clause in conjunction with the ON UPDATE clause.

   This statement creates a table that cascades updates of reference column values to the foreign key column, but if a reference column value is deleted, the corresponding foreign key values are set to NULL.

SQL

```
CREATE TABLE Orders (
    OrderID INT,
    OrderItem VARCHAR,
    OrderQuantity INT,
    CustomerNum INT,
    CONSTRAINT OrdersPK PRIMARY KEY (OrderID),
    CONSTRAINT CustomersFK FOREIGN KEY (CustomerNum) REFERENCES Customers (CustID)
        ON UPDATE CASCADE
        ON DELETE SET NULL)
```

- **CREATE TABLE** *table* **(..., CONSTRAINT** *fKeyName* **FOREIGN KEY (...) REFERENCES ... NOCHECK))** disables checking for referential integrity of the foreign key, meaning that an INSERT or UPDATE operation might specify a value for a foreign key column that does not correspond to a row in the reference table.

   The NOCHECK keyword also prevents the execution of the ON DELETE or ON UPDATE referential actions for the foreign key, if these actions are specified. The SQL query processor can use foreign keys to optimize joins among tables. However, if a foreign key is defined as NOCHECK, the SQL query processor does not consider it as defined. A NOCHECK foreign key is still reported to database driver catalog queries as a foreign key. For more information, see Using Foreign Keys.

Special Columns and Column Properties

Computed Columns

These syntaxes show how to define columns that are computed on INSERT or UPDATE rather than user-supplied. For more details on these columns, see Computing a column value on INSERT or UPDATE.

- **CREATE TABLE** *table* (*column type* **COMPUTECODE [OBJECTSCRIPT | PYTHON] {***code***}, ...)** defines a column in which values are computed and stored upon **INSERT** using the specified ObjectScript or Python code. If you omit the OBJECTSCRIPT or PYTHON keyword, the code defaults to ObjectScript. The values in computed columns remain unchanged by subsequent table updates, such as an **UPDATE** command or trigger code operations.

   This statement creates a table that, when a row is inserted, computes the Age column based on the date specified in the DOB column.

SQL/ObjectScript

```
CREATE TABLE MyStudents (
    Name VARCHAR(16) NOT NULL,
    DOB DATE,
    Age VARCHAR(12) COMPUTECODE {
        set bdate = $zdate({DOB}, 8)
        set today = $zdate($horolog,8)
        set {Age} = $select(bdate = "":"", 1:(today - bdate) \ 10000)},
    Grade INT)
```

SQL/Python

```
CREATE TABLE MyStudents (
    Name VARCHAR(16) NOT NULL,
    DOB DATE,
    Age VARCHAR(12) COMPUTECODE PYTHON {
        import datetime as d
        iris_date_offset = d.date(1840,12,31).toordinal()
        bdate = d.date.fromordinal(cols.getfield('DOB') + iris_date_offset).strftime("%Y%m%d")
        today = d.date.today().strftime("%Y%m%d")
        return str((int(today) - int(bdate)) // 10000) if bdate else ""},
    Grade INT)
```

- **CREATE TABLE** *table* (*column type* **COMPUTECODE ... {*code*} COMPUTEONCHANGE** (*column*, *column2*, ...), ...) recomputes the value of the computed column when any one of the table columns specified in the COMPUTEON-CHANGE clause changes in a subsequent table update. The recomputed value replaces the previously stored value. If a column specified in COMPUTEONCHANGE is not part of the table specification, then InterSystems SQL generates an SQLCODE -31 error.

  This statement recomputes the `Age` column when the `DOB` column is updated. The `Birthday` column includes the timestamp for when the column last changed.

SQL

```
CREATE TABLE MyStudents (
    Name VARCHAR(20) NOT NULL,
    DOB TIMESTAMP,
    Birthday VARCHAR(40) COMPUTECODE {
        set {Birthday} = $zdate({DOB})
        _" changed: "_$zdatetime($ztimestamp) }
        COMPUTEONCHANGE (DOB))
```

  COMPUTEONCHANGE defines the SqlComputeOnChange keyword with the %%UPDATE value for the class property corresponding to the column definition. This property value is initially computed as part of the INSERT operation and recomputed during an UPDATE operation. For a corresponding Persistent Class definition, see Defining a Table by Creating a Persistent Class.

- **CREATE TABLE** *table* (*column type* **COMPUTECODE ... {*code*} CALCULATED,** ...) specifies that the column value is not stored in the database but is instead generated each time the column is queried. Calculated columns reduce the size of the data storage but can slow query performance.

  This column defines the Calculated boolean keyword for the class property corresponding to the column definition. CALCULATED properties cannot be indexed unless the property is also SQLComputed.

  This statement calculates the value of the `DaysToBirthday` column, which changes depending on the current date. The `{*}` code is a shortcut syntax for specifying the column being computed, in this case `DaysToBirthday`.

SQL

```
CREATE TABLE MyStudents (
    Name VARCHAR(20) NOT NULL,
    DOB TIMESTAMP,
    DaysToBirthday INT COMPUTECODE {
        set {*} = $zdate({DOB},14) - $zdate($horolog,14) } CALCULATED)
```

- **CREATE TABLE** *table* (*column type* **COMPUTECODE ... {*code*} TRANSIENT,** ...) is similar to CALCULATED and also specifies that the column is not saved to the database.

This column defines the Transient boolean keyword for the class property corresponding to the column definition. TRANSIENT properties cannot be indexed.

The CALCULATED and TRANSIENT keywords are mutually exclusive and provide similar behavior. TRANSIENT means that InterSystems IRIS does not store the property. CALCULATED means that InterSystems IRIS does not allocate any instance memory for the property. Thus when CALCULATED is specified, TRANSIENT is implicitly set.

- **CREATE TABLE** *table* (*column type* **ON UPDATE** *updateSpec*, **...**) defines a column that is recomputed whenever a row is updated in the table, based on the value specified by *updateSpec*. You cannot specify an ON UPDATE clause if the column also has a COMPUTECODE data constraint.

  This statement creates a table containing a `LastUpdated` column whose values are updated to the current time any time the corresponding rows are updated. The timestamp values stored in the table have a precision of two digits.

```
CREATE TABLE MyStudents (
    Name VARCHAR(20) NOT NULL,
    DOB TIMESTAMP,
    LastUpdated TIMESTAMP DEFAULT CURRENT_TIMESTAMP(2) ON UPDATE CURRENT_TIMESTAMP(2))
```

- **CREATE TABLE** *table* (*column type* **IDENTITY, ...**) replaces the system-generated integer RowID column with the specified named column.

  Like the RowID column, this column behaves as a single-column IDKEY index whose values are unique system-generated integers, where each value serves as a unique record ID for the corresponding table row. Defining an IDENTITY column prevents the defining of the Primary Key as the IDKEY. You can define only one IDENTITY column per table. *type* must be an integer data type. If you omit *type*, then the data type is defined as BIGINT. The IDENTITY values cannot be user-specified and cannot be modified in an UPDATE statement.

  This statement sets the `IdNum` column as the IDKEY. This column is returned as part of selection queries such as `SELECT *`.

SQL

```
CREATE TABLE Employee (
    EmpNum INT NOT NULL,
    IdNum IDENTITY NOT NULL,
    Name CHAR(30) NOT NULL,
    CONSTRAINT EMPLOYEEPK PRIMARY KEY (EmpNum))
```

For more details about working with the IDENTITY column, see Creating Named RowId Column Using IDENTITY Keyword.

Counter Columns

InterSystems SQL provides three types of system-generated integer counter columns. These columns are not mutually exclusive and can be specified together within the same table. The data types of all three columns map to the %Library.BigInt data type class.

Counter Type	Scope of Counter	Automatically Incremented by	When Unsupplied value is	Unsupplied values	Duplicate Values	Columns of this type	Counter Reset by	Sharded Table Support
AUTO_INCREMENT	Per-table	INSERT	NULL or 0	Allowed, does not affect system counter	Allowed	One per table	TRUNCATE TABLE	Yes
SERIAL	Per-serial counter column	INSERT	NULL or 0	Allowed, may increment system counter	Allowed	Multiple per table	TRUNCATE TABLE	No
ROWVERSION	Namespace-wide	INSERT and UPDATE	Not applicable	Not allowed	Not allowed	One per table	Not reset	No

For more details on these counter columns, see RowVersion, AutoIncrement and Serial Counter Columns.

- **CREATE TABLE** *table* (*column type* **AUTO_INCREMENT, ...**) creates a counter column that increments upon each INSERT into the table. You can designate only one AUTO_INCREMENT counter column per table. You must set the AUTO_INCREMENT keyword after an explicit integer data type. For example:

```
CREATE TABLE MyStudents (
    Name VARCHAR(16) NOT NULL,
    DOB TIMESTAMP,
    AutoInc BIGINT AUTO_INCREMENT)
```

Alternatively, you can define an AUTO_INCREMENT column using the %Library.AutoIncrement data type. Thus the following are also valid column definition syntax: `MyAutoInc %AutoIncrement`, `MyAutoInc %AutoIncrement AUTO_INCREMENT`.

- **CREATE TABLE** *table* (*column* **SERIAL, ...**) creates a counter column that increments upon each INSERT into the table. You can designate multiple columns as SERIAL counter columns. Specify the SERIAL keyword in place of an explicit data type. For example:

SQL

```
CREATE TABLE MyStudents (
   Name VARCHAR(16) NOT NULL,
    DOB TIMESTAMP,
    Counter SERIAL)
```

- **CREATE TABLE** *table* (*column* **ROWVERSION, ...**) creates a counter column that increments upon each INSERT or UPDATE operation across all tables in the namespace. Specify the ROWVERSION keyword in place of an explicit data type. For example:

SQL

```
CREATE TABLE MyStudents (
    Name VARCHAR(16) NOT NULL,
    DOB TIMESTAMP,
    RowVer ROWVERSION)
```

%DESCRIPTION Keyword

- **CREATE TABLE** *table* **(..., %DESCRIPTION** *description***)** specifies a description for the table being created. Enclose the description text string in quotes. For example:

  SQL

  ```
  CREATE TABLE Employee (
      %Description 'Employees at XYZ Inc.',
      EmpNum INT PRIMARY KEY,
      NameLast VARCHAR(30) NOT NULL,
      NameFirst VARCHAR(30) NOT NULL,
      StartDate TIMESTAMP)
  ```

- **CREATE TABLE** *table* **(***column type* **%DESCRIPTION** *description***, ...)** specifies a description for a column. You can specify one description per column. Enclose the description text string in quotes. For example:

  SQL

  ```
  CREATE TABLE Employee (
      EmpNum INT PRIMARY KEY,
      NameLast VARCHAR(30) NOT NULL,
      NameFirst VARCHAR(30) NOT NULL,
      StartDate TIMESTAMP %Description 'Format: MM/DD/YYYY')
  ```

%PUBLICROWID Keyword

- **CREATE TABLE** *table* **(%PUBLICROWID, ...)** makes the unique, system-generated integer RowID column public. For example:

  SQL

  ```
  CREATE TABLE Employee (
      %PUBLICROWID,
      EmpNum INT PRIMARY KEY,
      NameLast VARCHAR(30) NOT NULL,
      NameFirst VARCHAR(30) NOT NULL,
      StartDate TIMESTAMP)
  ```

  This column is named "ID" and is assigned to column 1. The class corresponding to the table is defined with "Not SqlRowIdPrivate". **ALTER TABLE** cannot be used to specify %PUBLICROWID.

  If the RowID is public:

  – RowID values are displayed by SELECT *.

  – The RowID can be used as a foreign key reference.

  – If there is no defined primary key, the RowID is treated as an Implicit PRIMARY KEY constraint with the Constraint Name RowIDField_As_PKey.

  – The table cannot be used to copy data into a duplicate table without specifying the column names to be copied.

  For more details of the RowID column, see RowID Hidden?.

Collation Property

- **CREATE TABLE** *table* **(***column type* **COLLATE** *sqlCollation***, ...)** sets the collation type used to order and compare values in the specified column. Valid collation values are %EXACT, %MINUS, %PLUS, %SPACE, %SQLSTRING, %SQLUPPER, %TRUNCATE, and %MVR.

  This statement sorts the UserName column as a case-sensitive string, treating NULL and numeric values in the column as string characters.

SQL

```
CREATE TABLE Sample.People (
    UserName VARCHAR(30) COLLATE %SQLSTRING,
    FirstName VARCHAR(30),
    LastName VARCHAR(30))
```

Table Options

Sharded Tables

These syntaxes provide the option to define a *sharded table*, where table rows are automatically horizontally partitioned across data nodes by using one of its columns as a *shard key*. A sharded table improves the performance of queries against that table, especially for tables containing a large number of rows. To improve the performance of queries that join large tables, you can coshard the tables. *Cosharding* partitions rows in different tables that have matching shard key values into the same data nodes. For more details on sharding, see Horizontally Scaling for Data Volume with Sharding.

Note: Not all **CREATE TABLE** syntaxes support sharded tables. For more details, see Sharded Table Restrictions.

- **CREATE TABLE** *table* **... SHARD** defines a sharded table and uses the RowID column as the shard key. This is known as a *system-assigned shard key* (SASK). If the table has a defined IDENTITY column and no explicit shard key, InterSystems SQL uses the IDENTITY column as the SASK instead. Using a SASK is the simplest and most effective way to shard a table. Specify the SHARD keyword after the column definitions.

  This statement creates a sharded table that uses the default RowID column as the shard key.

  ```
  CREATE TABLE Vehicle (
      Make VARCHAR(30) NOT NULL,
      Model VARCHAR(20) NOT NULL,
      Year INT NOT NULL,
      Vin CHAR(17) NOT NULL)
  SHARD
  ```

- **CREATE TABLE** *table* **... SHARD KEY (**_shardKeyColumn_**, shardKeyColumn2, ...)** specifies a column, or comma-separated list of columns, to use as the shard key. This is known as a *user-defined shard key* (UDSK).

  This statement creates a table with a shard key composed of two columns.

  ```
  CREATE TABLE Car (
      Owner VARCHAR(30) NOT NULL,
      Plate VARCHAR(10) NOT NULL,
      State CHAR(2) NOT NULL)
  SHARD KEY (Plate, State)
  ```

  You can also use this syntax to coshard two or more tables that you are defining. Joins on the UDSK columns of cosharded tables perform much more efficiently than joins on non-UDSK columns, so defining UDSKs on the most frequently used set of join columns is recommended.

  These statements create two tables with defined shard keys on the columns Vin and VehicleNumber.

  ```
  CREATE TABLE Vehicle (
      Make VARCHAR(30) NOT NULL,
      Model VARCHAR(20) NOT NULL,
      Year INT NOT NULL
      Vin CHAR(17) NOT NULL)
  SHARD KEY (Vin)

  CREATE TABLE Citation (
      CitationID VARCHAR(8) NOT NULL,
      Date TIMESTAMP NOT NULL,
      LicenseNumber VARCHAR(12) NOT NULL,
      Plate VARCHAR(10) NOT NULL,
      VehicleNumber CHAR(17) NOT NULL)
  SHARD KEY (VehicleNumber)
  ```

These tables benefit from improved JOIN performance when joining on the UDSK column, such as this query that returns traffic citations associated with a vehicle:

SQL

```
SELECT * FROM Citation, Vehicle WHERE Citation.VehicleNumber = Vehicle.Vin
```

For more details on choosing a shard key, see Choose a Shard Key.

- **CREATE TABLE** *table* **... SHARD KEY (***coshardKeyColumn***) COSHARD WITH (***coshardTable***)** creates a table and sets one of its integer-valued columns as a shard key, *coshardKeyColumn*. You can use this shard key column in joins with another sharded table, *coshardTable*, which must have a system-assigned shard key (SASK) defined using the SHARD syntax. You can optionally define *coshardKeyColumn* as a foreign reference to the SASK column of *coshardTable* as well. The foreign key reference can be used to enforce referential integrity, while the shard key provides faster performance for joins with the other table by matching on its SASK column.

  This statement creates a table that defines the CustomerID column as the shard key. This key is used in cosharded joins with the shard key of an existing sharded table, Customer.

  ```
  CREATE TABLE Order (
      Date TIMESTAMP NOT NULL,
      Amount DECIMAL(10,2) NOT NULL,
      CustomerID CUSTOMER NOT NULL)
  SHARD KEY (CustomerID) COSHARD WITH Customer
  ```

Temporary Tables

- **CREATE GLOBAL TEMPORARY TABLE** *table* **...** creates the table as a global temporary table, where the table definition is available to all processes but the table data (including Stream data) and indexes persist only for the duration of the process that created the table. This data is stored in process-private globals and is deleted when the process terminates.

  This statement creates a temporary table:

  SQL

  ```
  CREATE GLOBAL TEMPORARY TABLE TempEmp (
      EmpNum INT NOT NULL,
      NameLast CHAR(30) NOT NULL,
      NameFirst CHAR(30) NOT NULL,
      CONSTRAINT EMPLOYEEPK PRIMARY KEY (EmpNum))
  ```

  Regardless of which process creates a temporary table, the owner of the temporary table is automatically set to _PUBLIC. This means that all users can access a cached temporary table definition. For example, if a stored procedure creates a temporary table, the table definition can be accessed by any user that is permitted to invoke the stored procedure. This applies only to the temporary table definition. The temporary table data is specific to the invocation, and therefore can only be accessed by the current user process.

  The table definition of a global temporary table is the same as a base table. A global temporary table must have a unique name. Attempting to give it the same name as an existing base table results in an SQLCODE -201 error. The table persists until it is explicitly deleted (using **DROP TABLE**). You can alter the table definition using **ALTER TABLE**.

  You can only define global temporary tables through DDL statements.

  Like standard InterSystems IRIS tables, the ClassType=persistent, and the class includes the Final keyword, indicating that it cannot have subclasses.

Table Storage

- **CREATE TABLE** *table* **... WITH STORAGETYPE = [ROW | COLUMNAR]** specifies the layout used to store the underlying data in the table.

  - Specify ROW to store data in rows. Row storage enables efficient transactions, such as when frequently updating or inserting rows in online transaction processing (OLTP) workflows. If you omit the WITH STORAGETYPE clause, the created table defaults to row storage.

  - Specify COLUMNAR to store data in columns. Columnar storage enables efficient queries, such as when filtering or aggregating data in specific columns in online analytical processing (OLAP) workflows. Columnar storage is an experimental feature for 2022.2. In previous InterSystems IRIS versions, all table data is stored in rows.

    Note: For performance reasons, unless the collation type is specified explicitly, columnar storage layouts default to using EXACT collation. Row storage layouts use namespace-default SQLUPPER collation.

  For more details on choosing a storage layout, see Choose an SQL Table Storage Layout.

  This statement creates a table with a columnar storage layout.

  ```
  CREATE TABLE Sample.TransactionHistory (
    AccountNumber INTEGER,
    TransactionDate DATE,
    Description VARCHAR(100),
    Amount NUMERIC(10,2),
    Type VARCHAR(10))
  WITH STORAGETYPE = COLUMNAR
  ```

  This statement creates a table with columnar storage by using the **CREATE TABLE AS SELECT** clause.

  ```
  CREATE TABLE Sample.TransactionHistory AS
    SELECT AccountNumber, TransactionDate, Description, Amount, Type
    FROM Sample.BankTransaction
  WITH STORAGETYPE = COLUMNAR
  ```

  Tip: You can use **CREATE TABLE AS SELECT** to experiment with creating tables that have different storage types and comparing their performance.

- **CREATE TABLE** *table* (*column type* **... WITH STORAGETYPE = [ROW | COLUMNAR], ...**) specifies individual columns as having row or columnar storage. All other columns default to row storage or to the storage type specified in the WITH STORAGETYPE clause at the end of the table definition.

  This statement creates a table with a columnar storage layout only for the Amount column. Because this statement omits the WITH STORAGETYPE clause at the end of the table definition, the rest of the columns default to a row storage layout.

  ```
  CREATE TABLE Sample.BankTransaction (
    AccountNumber INTEGER,
    TransactionDate DATE,
    Description VARCHAR(100),
    Amount NUMERIC(10,2) WITH STORAGETYPE = COLUMNAR,
    Type VARCHAR(10))
  ```

Class Parameters

- **CREATE TABLE** *table* **... WITH %CLASSPARAMETER** *pName = pValue*, **%CLASSPARAMETER** *pName2 = pValue2*, **...** specifies one or more %CLASSPARAMETER name-value pairs that define core aspects of the table being created. Each parameter name, *pName*, is set to the specified value, *pValue*.

In this statement, the USEEXTENTSET class parameter disables the use of generated Global names, such as ^EPgS.D8T6.1. These globals are used as IDKEY indexes into the data. The DEFAULTGLOBAL class parameter specifies ^GL.EMPLOYEE as an explicit Global name for indexes.

```
CREATE TABLE Employees (
    EmpNum INT NOT NULL,
    NameLast CHAR(30) NOT NULL,
    NameFirst CHAR(30) NOT NULL,
    CONSTRAINT EMPLOYEEPK PRIMARY KEY (EmpNum)
    )
WITH %CLASSPARAMETER USEEXTENTSET = 0,
     %CLASSPARAMETER DEFAULTGLOBAL = '^GL.EMPLOYEE'
```

You can use DEFAULTGLOBAL to specify an extended global reference, either the full reference (`%CLASSPARAMETER DEFAULTGLOBAL = '^|"USER"|GL.EMPLOYEE'`), or just the namespace portion (`%CLASSPARAMETER DEFAULTGLOBAL = '^|"USER"|'`).

Arguments

table

In a **CREATE TABLE** command, specify the name of the table you want to create as a valid identifier. A table name can be qualified or unqualified.

- An unqualified table name has the following syntax: `tablename`; it omits *schema* (and the period (.) character). An unqualified table name takes the default schema name. The initial system-wide default schema name is `SQLUser`, which corresponds to the default class package name `User`. Schema search path values are ignored.

  The system-wide default schema name can be configured.

  To determine the current system-wide default schema name, use the **$SYSTEM.SQL.Schema.Default()** method.

- A qualified table name has the following syntax: `schema.tablename`. It can specify either an existing schema name or a new schema name. Specifying an existing schema name places the table within that schema. Specifying a new schema name creates that schema (and associated class package) and places the table within that schema.

Table Name and Schema Name Conventions

Table names and schema names follow SQL identifier naming conventions, subject to additional constraints on the use of non-alphanumeric characters, uniqueness, and maximum length. Names beginning with a % character are reserved for system use. By default, schema names and table names are simple identifiers and are not case-sensitive.

InterSystems IRIS uses the table name to generate a corresponding class name. A class name contains only alphanumeric characters (letters and numbers) and must be unique within the first 96 characters. To generate a class name, InterSystems IRIS first strips out symbol (non-alphanumeric) characters from the table name, and then generates a unique class name, imposing uniqueness and maximum length restrictions.

InterSystems IRIS uses the schema name to generate a corresponding class package name. To generate a package name, first it either strips out or performs special processing of symbol (non-alphanumeric) characters in the schema name. InterSystems IRIS then generates a unique package name, imposing uniqueness and maximum length restrictions. A schema name is not case-sensitive but the corresponding class package name is. If you specify a schema name that differs only in case from an existing class package name, and the package definition is empty (contains no class definitions), InterSystems IRIS reconciles the two names by changing the case of the class package name.

You can use the same name for a schema and a table, but you cannot use the same name for a table and a view in the same schema. For more details on how package and class names are generated from schema and table names, see Table Names and Schema Names.

Table Name Character Restrictions

InterSystems IRIS supports 16-bit (wide) characters for table and column names. For most locales, accented letters can be used for table names and the accent marks are included in the generated class name.

Note: The Japanese locale does not support accented letter characters in identifiers. Japanese identifiers can contain (in addition to Japanese characters) the Latin letter characters A-Z and a-z (65–90 and 97–122), the underscore character (95), and the Greek capital letter characters (913–929 and 931–937). The `nls.Language` test uses [(the Contains operator) rather than = because there are different Japanese locales for different operating system platforms.

Check for Existing Tables

To determine if a table already exists in the current namespace, use **$SYSTEM.SQL.Schema.TableExists("schema.tname")**.

By default, when you try to create a table that has the same name as an existing table InterSystems IRIS rejects the create table attempt and issues an SQLCODE -201 error. To determine the current system-wide configuration setting, call **$SYSTEM.SQL.CurrentSettings()**, which displays an `Allow DDL CREATE TABLE or CREATE VIEW for existing table or view` setting. The default is 0, which is the recommended setting. If this option is set to 1, InterSystems IRIS deletes the class definition associated with the table and then recreates it. This is similar to performing a **DROP TABLE** to delete the existing table and then performing **CREATE TABLE**. In this case, it is strongly recommended that the **$SYSTEM.SQL.CurrentSettings()**, `Does DDL DROP TABLE delete the table's data?` value be set to 1 (the default). Refer to DROP TABLE for further details.

From the Management Portal, **System Administration**, **Configuration**, **SQL and Object Settings**, **SQL** you can set this option (and other similar create, alter, and drop options) system-wide by selecting the **Ignore redundant DDL statements** check box. For more information on configuring these settings, consult the SQL Configuration Parameters page.

The behavior of the predicate IF NOT EXISTS takes priority over the settings described above. These settings effectively overwrite the table and return SQLCODE 0. When IF NOT EXISTS is specified, the command does nothing and returns SQLCODE 1 along with a message.

column

In a **CREATE TABLE** command, specify the column name, or a comma-separated list of column names, used to define the columns of the table you are creating. You can specify the column names in any order, with a space separating the column name from its associated data type. For example: `CREATE TABLE myTable (column1 INT, column2 VARCHAR(10))`. By convention, each column definition is usually presented on a separate line and indentation is used. This is recommended for readability but is not required. Column names are also used to define unique, primary key, and foreign key constraints.

Enclose column name lists in parentheses.

Rather than defining a column, a column definition can reference an existing embedded serial object that defines multiple columns (properties). The column name is followed by the package and class name of the serial object. For example: `Office Sample.Address`. Do not specify a data type or data constraints, but you can specify a %DESCRIPTION. You cannot create an embedded serial object using **CREATE TABLE**.

Column Name Conventions

Column names follow identifier conventions, with the same naming restrictions as table names. Avoid beginning column names beginning with a % character, though column names beginning with %z or %Z are permitted. A column name should not exceed 128 characters. By default, column names are simple identifiers. They are not case-sensitive. Attempting to create a column name that differs only in letter case from another column in the same table generates an SQLCODE -306 error.

InterSystems IRIS uses the column name to generate a corresponding class property name. A property name contains only alphanumeric characters (letters and numbers) and is a maximum of 96 characters in length. To generate this property name, InterSystems IRIS first strips punctuation characters from the column name, and then generates a unique identifier of 96

or fewer characters. InterSystems IRIS substitutes an integer, beginning with 0, for the final character of a column name when this is needed to create a unique property name.

This example shows how InterSystems IRIS handles column names that differ only in punctuation. The corresponding class properties for these columns are named PatNum, PatNu0, and PatNu1:

SQL

```
CREATE TABLE MyPatients (
    _PatNum VARCHAR(16),
    %Pat@Num INTEGER,
    Pat_Num VARCHAR(30),
    CONSTRAINT Patient_PK PRIMARY KEY (_PatNum))
```

The column name, as specified in **CREATE TABLE**, is shown in the class property as the SqlFieldName keyword value.

During a dynamic **SELECT** operation, InterSystems IRIS might generate property name aliases to facilitate common letter case variants. For example, given the column name Home_Street, InterSystems IRIS might assign the property name aliases home_street, HOME_STREET, and HomeStreet. InterSystems IRIS does not assign an alias if that name would conflict with the name of another field name, or with an alias assigned to another field name.

type

The data type class of the column name specified by column. A specified data type limits a column's allowed data values to the values appropriate for that data type. InterSystems SQL supports most standard SQL data types.

You can specify either an InterSystems SQL data type (for example, VARCHAR(24) or CHARACTER VARYING(24)) or the class that the data type maps to (for example, %Library.String(MAXLEN=24) or %String(MAXLEN=24)).

Specify data type classes when you want to define additional data definition parameters, such as an enumerated list of permitted data values, pattern matching of permitted data values, maximum and minimum numeric values, and automatic truncation of data values that exceed the maximum length (MAXLEN).

Note: A data type class parameter default may differ from the InterSystems SQL data type default. For example, VARCHAR() and CHARACTER VARYING() default to MAXLEN=1; The corresponding data type class %Library.String defaults to MAXLEN=50.

InterSystems IRIS maps these standard SQL data types to InterSystems IRIS data types by providing an SQL.System-DataTypes mapping table and an SQL.UserDataTypes mapping table.

To view and modify the current data type mappings, go to the Management Portal, select **System Administration**, **Configuration**, **SQL and Object Settings**, **System DDL Mappings**. To create additional data type mappings, go to the Management Portal, select **System Administration**, **Configuration**, **SQL and Object Settings**, **User DDL Mappings**.

If you specify a data type in SQL for which no corresponding InterSystems IRIS data type exists, the SQL data type name is used as the data type for the corresponding class property. You must create this user-defined InterSystems IRIS data type before DDL runtime (SQLExecute).

You may also override data type mappings for a single parameter value. For instance, suppose you did not want VAR-CHAR(100) to map to the supplied standard mapping %String(MAXLEN=100). You could override this by added a DDL data type of 'VARCHAR(100)' to the table and then specify its corresponding InterSystems IRIS type. For example:

```
VARCHAR(100) maps to MyString100(MAXLEN=100)
```

Data Size

Following a data type, you can present the permissible data size in parentheses. Whitespace between the data type name and data size parentheses is permitted but not required.

For a string, data size represents the maximum number of characters. For example:

```
ProductName VARCHAR (64)
```

A numeric value that permits fractional numbers is represented as a pair of integers, (*p*,*s*). The first integer, *p*, is the data type precision. This number is not identical to numerical precision, that is, the number of digits in the number, because the underlying InterSystems IRIS data type classes do not have a precision. Instead, these classes use this number to calculate the MAXVAL and MINVAL values. The second integer, *s*, is the scale, which specifies the maximum number of decimal digits. For example:

```
UnitPrice NUMERIC(6,2)  /* maximum value 9999.99 */
```

For more details on how precision and scale work, see Data Types.

query

A SELECT query that supplies the column definitions and column data for a table being created using the **CREATE TABLE AS SELECT** syntax. This query can specify a table, a view, or multiple joined tables. However, it cannot contain any ? parameters like regular **SELECT** queries.

The data definition of the **CREATE TABLE AS SELECT** query is as follows:

- **CREATE TABLE AS SELECT** copies column definitions from the *query* table. To rename copied columns specify a column alias in the *query*.

  CREATE TABLE AS SELECT can copy column definitions from multiple tables if the *query* specifies joined tables.

- **CREATE TABLE AS SELECT** always defines the RowID as hidden.

  - If the source table has a hidden RowID, **CREATE TABLE AS SELECT** does not copy the source table RowID, but creates a new RowID column for the created table. Copied rows are assigned new sequential RowID values.

  - If the source table has a public (non-hidden) RowID, or if the query explicitly selects a hidden RowID, **CREATE TABLE AS SELECT** creates a new RowID column for the table. The source table RowID is copied into the new table as an ordinary BigInt column that is not hidden, not unique, and not required. If the source table RowID is named "ID", the new table's RowID is named "ID1".

- If the source table has an IDENTITY column, **CREATE TABLE AS SELECT** copies it and its current data as an ordinary BIGINT column for non-zero positive integers that is neither unique nor required.

- **CREATE TABLE AS SELECT** defines an IDKEY index. It does not copy indexes associated with copied column definitions.

- **CREATE TABLE AS SELECT** does not copy any column constraints: it does not copy NULL/NOT NULL, UNIQUE, Primary Key, or Foreign Key constraints associated with a copied column definition.

- **CREATE TABLE AS SELECT** does not copy a Default restriction or value associated with a copied column definition.

- **CREATE TABLE AS SELECT** does not copy a COMPUTECODE data constraint associated with a copied column definition.

- **CREATE TABLE AS SELECT** does not copy a %DESCRIPTION string associated with copied table or column definition.

defaultSpec

The default value of a column, specified in the DEFAULT clause as a literal value or as a keyword option. A string supplied as a literal default value must be enclosed in single quotes. A numeric default value does not require single quotes. For example:

SQL

```
CREATE TABLE membertest (
    MemberId INT NOT NULL,
    Membership_status CHAR(13) DEFAULT 'M',
    Membership_term INT DEFAULT 2)
```

The DEFAULT value is not validated when creating a table. When defined, a DEFAULT value can ignore data type, data length, and data constraint restrictions. However, when using **INSERT** to supply data to the table, the DEFAULT value is constrained. It is not limited by data type and data length restrictions, but *is* limited by data constraint restrictions. For example, a column defined `Ordernum INT UNIQUE DEFAULT 'No Number'` can take the default once, ignoring the INT data type restriction. However, this column cannot take the default a second time, as this would violate the UNIQUE column data constraint.

If no DEFAULT is specified, the implied default is NULL. If a column has a NOT NULL data constraint, you must specify a value for that column, either explicitly or by DEFAULT. Do not use the SQL zero-length string (empty string) as a NOT NULL default value. For more details on null values and the empty string, see NULL.

The DEFAULT data constraint accepts these keyword options: NULL, USER, CURRENT_USER, SESSION_USER, SYSTEM_USER, CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, SYSDATE, and OBJECTSCRIPT.

The USER, CURRENT_USER, and SESSION_USER default keywords set the column value to the ObjectScript $USERNAME special variable.

The CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, GETDATE, GETUTCDATE, and SYSDATE SQL functions can also be used as DEFAULT values. They are described in their respective reference pages. You can specify **CURRENT_TIME** or a timestamp function with or without a precision value when used as a DEFAULT value. If no precision is specified, InterSystems SQL uses the precision of the SQL configuration setting "Default time precision for GETDATE(), CURRENT_TIME, and CURRENT_TIMESTAMP", which defaults to 0. The DEFAULT function uses the time precision setting in effect when the **CREATE TABLE** statement is prepared and compiled, not at the time of statement execution.

CURRENT_TIMESTAMP can be specified as the default for a column of data type %Library.PosixTime or %Library.TimeStamp; the current date and time is stored in the format specified by the column's data type. CURRENT_TIMESTAMP, GETDATE, GETUTCDATE, and SYSDATE can be specified as a default for a %Library.TimeStamp column (data type TIMESTAMP or DATETIME). InterSystems IRIS converts the date value to the appropriate format for the data type.

SQL

```
CREATE TABLE mytest (
    TestId INT NOT NULL,
    CREATE_TIMESTAMP DATE DEFAULT CURRENT_TIMESTAMP(2),
    WORK_START TIMESTAMP DEFAULT SYSDATE)
```

You can use the TO_DATE function as the DEFAULT data constraint for data type DATE. You can use the TO_TIMESTAMP function as the DEFAULT data constraint for data type TIMESTAMP.

The OBJECTSCRIPT *literal* keyword phrase enables you to generate a default value by providing a quoted string containing ObjectScript code, as shown in the following example:

SQL

```
CREATE TABLE mytest (
    TestId INT NOT NULL,
    CREATE_DATE DATE DEFAULT OBJECTSCRIPT '+$HOROLOG' NOT NULL,
    LOGNUM NUMBER(12,0) DEFAULT OBJECTSCRIPT '$INCREMENT(^LogNumber)')
```

See the ObjectScript Reference for further information.

uniqueName

The name of the constraint listed in the CONSTRAINT UNIQUE clause, specified as a valid identifier. If specified as a delimited identifier, a constraint name can include the ".", "^", ",", and "->" characters. The constraint name uniquely identifies the constraint and is also used to derive the corresponding index name. This constraint name is required when using the ALTER TABLE command to drop a constraint from the table definition. Note that **ALTER TABLE** cannot drop a column that is listed in CONSTRAINT UNIQUE. Attempting to do so generates an SQLCODE -322 error.

The CONSTRAINT UNIQUE clause has this syntax:

```
CONSTRAINT uniqueName UNIQUE (column1,column2)
```

This constraint specifies that the combination of values of columns *column1* and *column2* must always be unique, even though either of these columns by itself may take non-unique values. You can specify one or more columns for this constraint.

All of the columns specified in this constraint must be defined in the column definition. Specifying a column that does not also appear in the column definitions generates an SQLCODE -86 error. The specified columns should be defined as NOT NULL. None of the specified columns should be defined as UNIQUE, as this would make specifying this constraint meaningless.

Columns can be specified in any order. The column order dictates the column order for the corresponding index definition. Duplicate column names are permitted. Although you may specify a single column name in the UNIQUE columns constraint, this would be functionally identical to specify the UNIQUE data constraint to that column. A single-column constraint does provide a constraint name for future use.

You can specify multiple unique column constraint statements in a table definition. Constraint statements can be specified anywhere in the column definition; by convention they are commonly placed at the end of the list of defined columns.

Refer to the Constraints option of Catalog Details for ways to list the columns of a table that are defined with a unique constraint.

pKeyName

The name of the primary key defined in the PRIMARY KEY constraint clause, specified as a valid identifier. If specified as a delimited identifier, a constraint name can include the ".", "^", ",", and "->" characters. This optional constraint name is used in ALTER TABLE to identify a defined constraint.

For more details on defining the primary key of a table, see Defining a Primary Key.

fKeyName

The name of a foreign key defined in the FOREIGN KEY constraint clause, specified as a valid identifier. If specified as a delimited identifier, a constraint name can include the ".", "^", ",", and "->" characters. This optional constraint name is used in ALTER TABLE to identify a defined constraint.

For more details on defining a foreign key in a table, see Defining a Foreign Key.

refTable

The name of the table to reference in the FOREIGN KEY clause, specified as a valid identifier. A table name can be qualified (schema.table), or unqualified (table).

refColumn

A column name or a comma-separated list of existing column names defined in the reference table that is specified in the foreign key constraint. Enclose the referenced columns in parentheses. If you omit *refColumn*, then **CREATE TABLE** assigns a default reference column, as described in Defining a Foreign Key.

To specify an explicit RowID as the reference column, specify *refColumn* as %ID. For example: FOREIGN KEY (CustomerNum) REFERENCES Customers (%ID). This value is synonymous with an omitted column name, provided

that the reference table has no primary key or foreign key specified. If the class definition for the table contains SqlRowId-Name, you can specify this value as the explicit RowID.

refAction

If a table contains a foreign key, a change to one table has an effect on another table. To keep the data consistent, when you define a foreign key, you also define what effect a change to the record from which the foreign key data comes has on the foreign key value. In **CREATE TABLE**, the ON DELETE *refAction* and ON UPDATE *refAction* clauses specify what action to take when a foreign key column specified by refColumn is changed.

- The ON DELETE clause defines the DELETE rule for the reference table. When an attempt to delete a row from the reference table is made, the ON DELETE clause defines what action to take for the rows in the reference table.

- The ON UPDATE clause defines the UPDATE rule for the reference table. When an attempt to change (update) the primary key value of a row from the reference table is made, the ON UPDATE clause defines what action to take for the rows in the reference table.

InterSystems SQL supports these foreign key referential actions:

Referential Action	Description
NO ACTION (default)	If any row in the foreign key column references the row being deleted or updated, the delete or update fails. This constraint does not apply if the foreign key references itself.
SET DEFAULT	Set the foreign key columns that reference the row being deleted or updated to their default values. If the foreign key column does not have a default value, it is set to NULL. A row must exist in the referenced table that contains an entry for the default value.
SET NULL	Set the foreign key columns that reference the row being deleted or updated to NULL. The foreign key columns must allow NULL values.
CASCADE	ON DELETE — Also delete the rows of the foreign key columns that reference the row being deleted. ON UPDATE — Also update the rows of foreign key columns that reference the row being updated.

Do not define two foreign keys with different names that reference the same column combination and perform contradictory referential actions. In accordance with the ANSI standard, InterSystems SQL does not issue an error if such cases (for example, ON DELETE CASCADE and ON DELETE SET NULL). Instead, InterSystems SQL issues an error when a **DELETE** or **UPDATE** operation encounters these contradictory foreign key definitions. For more information, see Using Foreign Keys.

code

Lines of code used in the COMPUTECODE data constraint to compute a default value of a column. Specify the code in curly braces. Whitespace and line returns are permitted before or after the curly braces.

The programming language of the code depends on the value you set in the COMPUTECODE clause:

- COMPUTECODE or COMPUTECODE OBJECTSCRIPT — Specify *code* as ObjectScript code. Within the code, you can reference SQL column names with curly brace delimiters, for example, {DOB}. The ObjectScript code can

contain Embedded SQL. In the projected class, COMPUTECODE specifies the SqlComputeCode column name and the computation for its value.

- COMPUTECODE PYTHON — Specify *code* as Python code. Within the code, you can reference SQL column names by using the `cols.getfield` method, for example, `cols.getfield('DOB')`. In the projected class, COMPUTE-CODE specifies the *Property*`Computation` class method, which stores the code that computes the column values. *Property* is the name of the column being computed. The projected class uses this class method in place of a `SqlComputeCode` property keyword.

When you specify a computed field name, either in COMPUTECODE or in the `SqlComputeCode` property keyword, you must specify the SQL field name, not the corresponding generated table property name.

A default data value supplied by COMPUTECODE must be in Logical (internal storage) mode. Embedded SQL in compute code is automatically compiled and run in Logical mode.

The following example defines the Birthday COMPUTECODE column. It uses ObjectScript code to compute its default value from the DOB column value:

SQL

```
CREATE TABLE MyStudents (
    Name VARCHAR(16) NOT NULL,
    DOB TIMESTAMP,
    Birthday VARCHAR(12) COMPUTECODE {SET {Birthday}=$PIECE($ZDATE({DOB}),9),",")},
    Grade INT)
```

The COMPUTECODE can contain the pseudo-field reference variables %%CLASSNAME, %%CLASSNAMEQ, %%OPERATION, %%TABLENAME, and %%ID. These variables are translated into specific values at class compilation time. The variables are not case-sensitive.

- In ObjectScript compute code, call pseudo-field reference variables by enclosing them in curly braces. For example: `{%%CLASSNAME}`

- In Python compute code, call pseudo-field reference variables by using the `cols.getfield` method. For example: `cols.getfield(%%CLASSNAME)`

The COMPUTECODE value is a default. It is returned only if you did not supply a value to the column. The COMPUTE-CODE value is not limited by data type restrictions. The COMPUTECODE value *is* limited by the UNIQUE data constraint and other data constraint restrictions. If you specify both a DEFAULT and a COMPUTECODE, the DEFAULT is always taken.

COMPUTECODE can optionally take a COMPUTEONCHANGE, CALCULATED, or TRANSIENT keyword.

If an error in the ObjectScript COMPUTECODE code occurs, SQL does not detect this error until the code is executed for the first time. Therefore, if the value is first computed upon insert, the INSERT operation fails with an SQLCODE -415 error; if the value is first computed upon update, the UPDATE operation fails with an SQLCODE -415 error; if the value is first computed when queried, the SELECT operation fails with an SQLCODE -350 error.

A COMPUTECODE stored value can be indexed. The application developer is responsible for making sure that computed column stored values are validated and normalized (numbers in canonical form), based on their data type, especially if you define (or intend to define) an index for the computed column.

updateSpec

When you create a table and specify a column using the ON UPDATE clause, that column is computed every time a row is updated in the table. The most common use of this feature is to define a column in a table that contains a timestamp value for the last time the row was updated.

Available *updateSpec* options are:

```
CURRENT_DATE | CURRENT_TIME[(precision)] | CURRENT_TIMESTAMP[(precision)] | GETDATE([prec]) |
GETUTCDATE([prec]) | SYSDATE |  USER | CURRENT_USER | SESSION_USER | SYSTEM_USER |  NULL | <literal>
| -<number>
```

The following example sets the RowTS column to the current timestamp value when a row is inserted and each time that row is updated:

```
CREATE TABLE mytest (
    Name VARCHAR(48),
    RowTS TIMESTAMP DEFAULT Current_Timestamp(6) ON UPDATE Current_Timestamp(6) )
```

In this example, the DEFAULT keyword sets RowTS to the current timestamp on INSERT if no explicit value is specified for the RowTS column. If an UPDATE specifies an explicit value for the RowTS column, the ON UPDATE keyword validates, but ignores, the specified value, and updates RowTS with the current timestamp. If the specified value fails validation, a SQLCODE -105 error is generated.

The following example sets the `HasBeenUpdated` column to a boolean value:

```
CREATE TABLE mytest (
    Name VARCHAR(48),
    HasBeenUpdated TINYINT DEFAULT 0 ON UPDATE 1 )
```

The following example sets the `WhoLastUpdated` column to the current user name:

```
CREATE TABLE mytest (
    Name VARCHAR(48),
    WhoLastUpdated VARCHAR(48) DEFAULT CURRENT_USER ON UPDATE CURRENT_USER )
```

You cannot specify an ON UPDATE clause if the column also has a COMPUTECODE data constraint. Attempting to do so results in an SQLCODE -1 error at compile or prepare time.

description

InterSystems SQL provides a %DESCRIPTION keyword, which you can use to provide a description for documenting a table or a column. %DESCRIPTION is followed by text string, *description*, enclosed in single quotes. This text can be of any length, and can contain any characters, including blank spaces. (A single-quote character within a description is represented by two single quotes. For example: `'Joe''s Table'`.) A table can have a %DESCRIPTION. Each column of a table can have its own %DESCRIPTION, specified after the data type. If you specify more than one table-wide %DESCRIPTION for a table, InterSystems IRIS issues an SQLCODE -82 error. If you specify more than one %DESCRIPTION for a column, the system retains only the last %DESCRIPTION specified. You cannot use **ALTER TABLE** to alter existing descriptions.

In the corresponding persistent class definition, a description appears prefaced by three slashes on the line immediately before the corresponding class (table) or property (column) syntax. For example: `/// Joe's Table`. In the *Class Reference* for the corresponding persistent class, the table description appears at the top just after the class name and SQL table name; a column description appears just after the corresponding property syntax.

You can display %DESCRIPTION text using the DESCRIPTION property of INFORMATION.SCHEMA.TABLES or INFORMATION.SCHEMA.COLUMNS. For example:

SQL

```
SELECT COLUMN_NAME,DESCRIPTION FROM INFORMATION_SCHEMA.COLUMNS WHERE TABLE_NAME='MyTable'
```

sqlCollation

The type of collation used to sort values of a column, specified as one of the following SQL collation types: %EXACT, %MINUS, %PLUS, %SPACE, %SQLSTRING, %SQLUPPER, %TRUNCATE, or %MVR. Collation keywords are not case-sensitive. It is recommended that you specify the optional keyword COLLATE before the collation parameter for

---

programming clarity, but this keyword is not required. The percent sign (%) prefix to the various collation parameter keywords is also optional.

The default is the namespace default collation (%SQLUPPER, unless changed). %SQLSTRING, %SQLUPPER, and %TRUNCATE may be specified with an optional maximum length truncation argument, an integer enclosed in parentheses. For more information on collation, see Table Field/Property Definition Collation.

%EXACT collation follows the ANSI (or Unicode) character collation sequence. This provides case-sensitive string collation and recognizes leading and trailing blanks and tab characters.

The %SQLUPPER collation converts all letters to uppercase for the purpose of collation. For further details on not case-sensitive collation, refer to the %SQLUPPER function.

The %SPACE and %SQLUPPER collations append a blank space to the data. This forces string collation of NULL and numeric values.

The %SQLSTRING, %SQLUPPER, and %TRUNCATE collations provide an optional *maxlen* parameter, which must be enclosed in parentheses. *maxlen* is a truncation integer that specifies the maximum number of characters to consider when performing collation. This parameter is useful when creating indexes with columns containing large data values.

The %PLUS and %MINUS collations handle NULL as a zero (0) value.

InterSystems SQL provides functions for most of these collation types. Refer to the %EXACT, %SQLSTRING, %SQLUPPER, %TRUNCATE functions for further details.

ObjectScript provides the **Collation()** method of the %SYSTEM.Util class for data collation conversion.

Note: To change the namespace default collation from %SQLUPPER (which is not case-sensitive) to another collation type, such as %SQLSTRING (which is case-sensitive), use the following command:

ObjectScript

```
WRITE $$SetEnvironment^%apiOBJ("collation","%Library.String","SQLSTRING")
```

After issuing this command, you must purge indexes, recompile all classes, and then rebuild indexes. Do not rebuild indexes while the table's data is being accessed by other users. Doing so may result in inaccurate query results.

shardKeyColumn

The column, or comma-separated list of columns, used as the shard key. Specify *shardKeyColumn* in the SHARD KEY clause, immediately after the closing parenthesis of the table column list but before the WITH clause (if specified). Specifying the shard key definition as an element within the table column list is supported for backwards compatibility, but defining a shard key in both locations generates an SQLCODE -327 error.

You cannot define the RowID column as the shard key. However, if the created table includes an IDENTITY column or IDKEY, you can define either of those columns as the shard key.

For information on choosing a shard key, see Choose a Shard Key.

coshardKeyColumn

The name of the shard key column that is used in cosharded joins with the shard key of the table defined in *coshardTable*. Specify *coshardKeyColumn* in the COSHARD WITH syntax: SHARD KEY (*coshardKeyColumn*) COSHARD WITH *coshardTable*.

coshardTable

The name of an existing table that the table being created coshards with. The table specified in the COSHARD WITH clause must be a sharded table with a system-assigned shard key.

When you specify this table, InterSystems IRIS sets the CoshardWith index keyword in the ShardKey index for the sharded table. This CoshardWith index keyword is equal to the class that projects the table.

To determine which sharded tables specified in a query are cosharded, view the Cosharding comment option.

pName = pValue

A %CLASSPARAMETER name-value pair that sets the class parameter named *pName* to the value *pValue*. You can specify multiple %CLASSPARAMETER clauses using comma-separated name-value pairs. For example: `WITH %CLASSPARAMETER DEFAULTGLOBAL = '^GL.EMPLOYEE', %CLASSPARAMETER MANAGEDEXTENT 0`. Separate the name and value using an equal sign or at least one space. Class parameter values are literal strings and numbers and must be defined as constant values.

Some of the class parameters currently in use are: ALLOWIDENTITYINSERT, *DATALOCATIONGLOBAL*, *DEFAULTGLOBAL, DSINTERVAL, DSTIME, EXTENTQUERYSPEC, EXTENTSIZE, GUIDENABLED, MANAGEDEXTENT, READONLY, ROWLEVELSECURITY, SQLPREVENTFULLSCAN, USEEXTENTSET, VERSIONCLIENTNAME, VERSIONPROPERTY*. Refer to the %Library.Persistent class for descriptions of these class parameters.

You can use the *USEEXTENTSET* and *DEFAULTGLOBAL* class parameters to define the global naming strategy for table data storage and index data storage.

The *IDENTIFIEDBY* class parameter is deprecated. You must convert IDENTIFIEDBY relationships to proper Parent/Child relationships to be supported in InterSystems IRIS.

A **CREATE TABLE** that defines a sharded table cannot define the *DEFAULTGLOBAL, DSINTERVAL, DSTIME*, or *VERSIONPROPERTY* class parameter.

You can specify additional class parameters as needed. For more details, see Class Parameters.

Examples

Create and Populate Table

Use **CREATE TABLE** to create a table, `Employee`, with several columns:

- The `EmpNum` column (containing the employee's company ID number) is an integer value that cannot be NULL; additionally, it is declared as a primary key for the table and automatically increments each time a row is inserted into the table.

- The employee's last and first names are stored in character string columns that have a maximum length of 30 and cannot be NULL.

- The remaining columns are for the employee's start date, accrued vacation time, and accrued sick time, which use the TIMESTAMP and INT data types.

```
CREATE TABLE Employee (
    EmpNum INT NOT NULL AUTO_INCREMENT,
    NameLast CHAR(30) NOT NULL,
    NameFirst CHAR(30) NOT NULL,
    StartDate TIMESTAMP,
    AccruedVacation INT,
    AccruedSickLeave INT,
    CONSTRAINT EMPLOYEEPK PRIMARY KEY (EmpNum))
```

To modify the table schema, use ALTER TABLE. For example, this statement changes the name of the table from `Employee` to `Employees`.

```
ALTER TABLE Employee RENAME Employees
```

To insert rows into a table, use INSERT. For example, this statement inserts a row with only the required columns in the table. The `EmpNum` column is also required, but you do not need to specify it because it auto-increments.

SQL

```
INSERT INTO Employees (NameLast, NameFirst) VALUES ('Zubik','Jules')
```

To update inserted rows, use UPDATE. For example, in the inserted row, this statement sets a value in one of the columns that was missing data.

SQL

```
UPDATE Employees SET AccruedVacation = 15 WHERE Employees.EmpNum = 1
```

To delete a row, use DELETE. For example, this statement deletes the inserted row.

SQL

```
DELETE FROM Employess WHERE EmpNum = 1
```

To delete an entire table, use DROP TABLE. Be careful using **DROP TABLE**. Unless you specify the %NODELDATA keyword, this command deletes both the table and all associated data.

SQL

```
DROP TABLE Employess
```

Security and Privileges

The **CREATE TABLE** command is a privileged operation that requires %CREATE_TABLE administrative privileges. Executing a **CREATE TABLE** command without these privileges results in an SQLCODE -99 error. To assign %CRE-ATE_TABLE privileges to a user or role, use the GRANT command, assuming that you hold appropriate granting privileges. If you are using the **CREATE TABLE AS SELECT** syntax, then you must have **SELECT** privilege on the table specified in the *query*. Administrative privileges are namespace-specific. For more details, see Privileges.

By default, **CREATE TABLE** security privileges are enforced. To configure this privilege requirement system-wide, use the **$SYSTEM.SQL.Util.SetOption()** method. For example: SET status=$SYSTEM.SQL.Util.SetOption("SQLSecurity",0,.oldval). To determine the current setting, call the **$SYSTEM.SQL.CurrentSettings()** method, which displays an SQL security enabled setting. The default is 1 (enabled). When SQL security is enabled (recommended), a user can perform actions only on table or views for which they have privileges. Set this method to 0 to disable SQL security for any new process started after changing this setting. This means that privilege-based table/view security is suppressed. You can create a table without specifying a user. In this case, Dynamic SQL assigns "_SYSTEM" as user, and Embedded SQL assigns "" (the empty string) as user. Any user can perform actions on a table or view even if that user has no privileges to do so.

Embedded SQL does not use SQL privileges. In Embedded SQL, you can use the **$SYSTEM.Security.Login()** method to log in as a user with appropriate privileges. You must have the **%Service_Login:Use** privilege to invoke the **$SYSTEM.Security.Login()** method. For example:

ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
NEW SQLCODE,%msg
&sql(CREATE TABLE MyTable (col1 INT, col2 INT))
IF SQLCODE=0 {WRITE !,"Table created"}
ELSE {WRITE !,"SQLCODE=",SQLCODE,": ",%msg }
```

For more information, see %SYSTEM.Security.

If CREATE TABLE is used with computed columns that require executing code, the user will need %Development:USE privileges in addition to %CREATE_TABLE privileges unless the command is used in Embedded SQL.

Users can also avoid privilege checks by creating a command with the %SQL.Statement class and using either the %Prepare() method with the checkPriv argument set to 0 or the %ExecDirectNoPriv() method.

More About

Class Definitions of Created Tables

When you create an SQL table using **CREATE TABLE**, InterSystems IRIS® automatically creates a persistent class corresponding to this table definition, with properties corresponding to the column definitions.

CREATE TABLE defines the corresponding class as DdlAllowed. It does not specify an explicit StorageStrategy in the corresponding class definition; it uses the default storage %Storage.Persistent. By default, **CREATE TABLE** specifies the Final class keyword in the corresponding class definition, indicating that it cannot have subclasses. (The default is 1; you can change this default system-wide using the **$SYSTEM.SQL.Util.SetOption()** method SET status=$SYSTEM.SQL.Util.SetOption("DDLFinal",0,.oldval); to determine the current setting, call the **$SYSTEM.SQL.CurrentSettings()** method).

Defining a Primary Key

Defining a primary key is optional. When you define a table, InterSystems IRIS automatically creates a generated column, the RowID Column (default name "ID") which functions as a unique row identifier. As each record is added to a table, InterSystems IRIS assigns a unique non-modifiable positive integer to that record's RowID column. You can optionally define a primary key that also functions as a unique row identifier. A primary key allows the user to define a row identifier that is meaningful to the application. For example, a primary key might be an Employee ID column, a Social Security Number, a Patient Record ID column, or an inventory stock number. You can explicitly define a column or group of columns as the primary record identifier by using the PRIMARY KEY clause.

A primary key accepts only unique values and does not accept NULL. (The primary key index property is not automatically defined as Required; however, it effectively is required, since a NULL value cannot be filed or saved for a primary key column.) The collation type of a primary key is specified in the definition of the column itself.

Refer to the Constraints option of Catalog Details for ways to list the columns of a table that are defined as the primary key.

For more details, see Primary Key.

Primary Key As IDKEY

By default, the primary key is *not* the unique IDKEY index. In many cases this is preferable, because it enables you to update primary key values, set the collation type for the primary key, and so on. There are cases where it is preferable to define the primary key as the IDKEY index. Be aware that this imposes the IDKEY restrictions on the future use of the primary key.

If you add a primary key constraint to an existing column, the column may also be automatically defined as an IDKEY index. This depends on whether data is present and upon a configuration setting established in one of the following ways:

- The SQL SET OPTION PKEY_IS_IDKEY statement.

- The system-wide **$SYSTEM.SQL.Util.SetOption()** method configuration option DDLPKeyNotIDKey. To determine the current setting, call **$SYSTEM.SQL.CurrentSettings()** which displays Are primary keys created through DDL not ID keys; the default is 1.

- Go to the Management Portal, select **System Administration**, **Configuration**, **SQL and Object Settings**, **SQL**. View the current setting of **Define primary key as ID key for tables created via DDL**.

  - If the check box is not selected (the default), the Primary Key does not become the IDKEY index in the class definition. Access to records using a primary key that is not the IDKEY is significantly less efficient; however, this type of primary key value can be modified.

– If the check box is selected, when a Primary Key constraint is specified through DDL, it automatically becomes the IDKEY index in the class definition. With this option selected, data access is more efficient, but a primary key value, once set, can never be modified.

However, if an IDENTITY column is defined in the table, the primary key can never be defined as the IDKEY, even when you have used one of these configuration setting to define the primary key as the IDKEY.

InterSystems IRIS supports properties (columns) that are part of the IDKEY index to be SqlComputed. For example, a parent reference column. The property must be a triggered computed column. An IDKEY property defined as SqlComputed is only computed upon the initial save of a new Object or an **INSERT** operation. **UPDATE** computation is not supported, because columns that are part of the IDKEY index cannot be updated.

No Primary Key

In most cases, you should explicitly define a primary key. However, if a primary key is not designated, InterSystems IRIS attempts to use another column as the primary key for ODBC/JDBC projection, according to the following rules:

1. If there is an IDKEY index on a *single* column, report the IDKEY column as the SQLPrimaryKey column.

2. Else if the class is defined with SqlRowIdPrivate=0 (the default), report the RowID column as the SQLPrimaryKey column.

3. Else if there is an IDKEY index, report the IDKEY columns as the SQLPrimaryKey columns.

4. Else do not report an SQLPrimaryKey.

Multiple Primary Keys

You can only define one primary key. By default, InterSystems IRIS rejects an attempt to define a primary key when one already exists, or to define the same primary key twice, and issues an SQLCODE -307 error. The SQLCODE -307 error is issued even if the second definition of the primary key is identical to the first definition. To determine the current configuration, call **$SYSTEM.SQL.CurrentSettings()**, which displays an `Allow create primary key through DDL when key exists` setting. The default is 0 (No), which is the recommended configuration setting. If this option is set to 1 (Yes), InterSystems IRIS drops the existing primary key constraint and establishes the last-specified primary key as the table's primary key.

From the Management Portal, **System Administration**, **Configuration**, **SQL and Object Settings**, **SQL** you can set this option (and other similar create, alter, and drop options) system-wide by selecting the **Ignore redundant DDL statements** check box.

For example, the following **CREATE TABLE** statement:

SQL

```
CREATE TABLE MyTable (f1 VARCHAR(16),
CONSTRAINT MyTablePK PRIMARY KEY (f1))
```

creates the primary key (if none exists). A subsequent **ALTER TABLE** statement:

SQL

```
ALTER TABLE MyTable ADD CONSTRAINT MyTablePK PRIMARY KEY (f1)
```

generates an SQLCODE -307 error.

Defining a Foreign Key

A foreign key is a column that references another table; the value stored in the foreign key column is a value that uniquely identifies a record in the other table. The simplest form of this reference is shown in the following example, in which the foreign key explicitly references the primary key column CustID in the Customers table:

SQL

```
CREATE TABLE Orders (
    OrderID INT UNIQUE NOT NULL,
    OrderItem VARCHAR,
    OrderQuantity INT,
    CustomerNum INT,
    CONSTRAINT OrdersPK PRIMARY KEY (OrderID),
    CONSTRAINT CustomersFK FOREIGN KEY (CustomerNum) REFERENCES Customers (CustID))
```

Most commonly, a foreign key references the primary key column of the other table. However, a foreign key can reference a RowID (ID) or an IDENTITY column. In every case, the foreign key reference must exist in the referenced table and must be defined as unique; the referenced column cannot contain duplicate values or NULL.

In a foreign key definition, you can specify:

- One column name: FOREIGN KEY (CustomerNum) REFERENCES Customers (CustID). The foreign key column (CustomerNum) and referenced column (CustID) may have different names (or the same name), but must have the same data type and column constraints.

- A comma-separated list of column names: FOREIGN KEY (CustomerNum,SalespersonNum) REFERENCES Customers (CustID,SalespID). The foreign key columns and referenced columns must correspond in number of columns and in order listed.

- An omitted column name: FOREIGN KEY (CustomerNum) REFERENCES Customers.

- An explicit RowID column: FOREIGN KEY (CustomerNum) REFERENCES Customers (%ID). Synonymous with an omitted column name. If the class definition for the table contains SqlRowIdName you can specify this value as the explicit RowID.

If you define a foreign key and omit the referenced column name, the foreign key defaults as follows:

1. The primary key column defined for the specified table.

2. If the specified table does not have a defined primary key, the foreign key defaults to the IDENTITY column defined for the specified table.

3. If the specified table does not have either a defined primary key or a defined IDENTITY column, the foreign key defaults to the RowID. This occurs only if the specified table defines the RowID as public; the specified table definition can do this explicitly, either by specifying the %PUBLICROWID keyword, or through the corresponding class definition with SqlRowIdPrivate=0 (the default). If the specified table does not define the RowID as public, InterSystems IRIS issues an SQLCODE -315 error. You must omit the referenced column name when defining a foreign key on the RowID; attempting to explicitly specify ID as the referenced column name results in an SQLCODE -316 error.

If none of these defaults apply, InterSystems IRIS issues an SQLCODE -315 error.

Refer to the Constraints option of Catalog Details for ways to list the columns of a table that are defined as foreign key columns and the generated Constraint Name for a foreign key.

In a class definition, you can specify a Foreign Key that contains a column based on a parent table IDKEY property, as shown in the following example:

```
ForeignKey Claim(CheckWriterPost.Hmo,Id,Claim) References SQLUser.Claim.Claim(DBMSKeyIndex);
```

Because the parent column defined in a foreign key of a child has to be part of the IDKEY index of the parent class, the only referential action supported for foreign keys of this type is NO ACTION.

- If a foreign key references a nonexistent table, InterSystems IRIS issues an SQLCODE -310 error, with additional information provided in %msg.

- If a foreign key references a nonexistent column, InterSystems IRIS issues an SQLCODE -316 error, with additional information provided in %msg.

- If a foreign key references a nonunique column, InterSystems IRIS issues an SQLCODE -314 error, with additional information provided in %msg.

If the foreign key column references a single column, the two columns must have the same data type and column data constraints.

In a parent/child relationship, there is no defined ordering of the children. Application code must not rely on any particular ordering.

You can define a foreign key constraint that references a class in a database that is mounted read-only. To define a FOREIGN KEY, the user must have REFERENCES privilege on the table being referenced or on the columns of the table being referenced. REFERENCES privilege is required if the **CREATE TABLE** is executed via Dynamic SQL or a database driver.

Sharded Tables and Foreign Keys

Foreign keys are supported for any combination of sharded and unsharded tables, including: key table sharded, fkey table unsharded; key table unsharded, fkey table sharded; and both key table and fkey table sharded. The key in the referenced table can be the shard key or another key. A foreign key can be a single column or multiple columns.

NO ACTION is the only referential action supported for sharded tables.

For more details, see Querying the Sharded Cluster.

Implicit Foreign Key

It is preferable to explicitly define all foreign keys. If there is an explicit foreign key defined, InterSystems IRIS reports this constraint and the implicit foreign key constraint is not defined.

However, it is possible to project implicit foreign keys to ODBC/JDBC and the Management Portal. These implicit foreign keys are reported as UPDATE and DELETE referential actions of NO ACTION. This implicit reference foreign key is not a true foreign key as there are no referential actions enforced. The name of this foreign key reported for the reference is "IMPLICIT_FKEY_REFERENCE__"_columnname. The reporting of this reference as a foreign key is provided for interoperability with third-party tools.

Bitmap Extent Index

When you create a table using **CREATE TABLE**, by default InterSystems IRIS automatically defines a bitmap extent index for the corresponding class. The SQL MapName of the bitmap extent index is %%DDLBEIndex:

```
Index DDLBEIndex [ Extent, SqlName = "%%DDLBEIndex", Type = bitmap ];
```

This bitmap extent index is *not* created in any of the following circumstances:

- The table is defined as a global temporary table (CREATE TABLE GLOBAL TEMPORARY TABLE ...).

- The table defines an explicit IDKEY index.

- The table contains a defined IDENTITY column that does not have MINVAL=1.

- The **$SYSTEM.SQL.Util.SetOption()** method `DDLDefineBitmapExtent` option is set to 0 to override the default system-wide. To determine the current setting, call the **$SYSTEM.SQL.CurrentSettings()** method, which displays a `Do classes created by a DDL CREATE TABLE statement define a bitmap extent index` setting.

If, after creating a bitmap index, the CREATE BITMAPEXTENT INDEX command is run against a table where a bitmap extent index was automatically defined, the bitmap extent index previously defined is renamed to the name specified by the CREATE BITMAPEXTENT INDEX statement.

For DDL operations that automatically delete an existing bitmap extent index, refer to ALTER TABLE.

For more details, see Bitmap Extent Index.

Creating Named RowId Column Using IDENTITY Keyword

InterSystems SQL automatically creates a RowID column for each table, which contains a system-generated integer that serves as a unique record id. The optional IDENTITY keyword allows you to define a named column with the same properties as a RowID record id column. An IDENTITY column behaves as a single-column IDKEY index, whose value is a unique system-generated integer.

Defining an IDENTITY column prevents the defining of the Primary Key as the IDKEY.

Just as with any system-generated ID column, an IDENTITY column has the following characteristics:

- You can only define one column per table as an IDENTITY column. Attempting to define more than one IDENTITY column for a table generates an SQLCODE -308 error.

- The data type of an IDENTITY column must be an integer data type. If you do not specify a data type, its data type is automatically defined as BIGINT. You can specify any integer data type, such as INTEGER or SMALLINT; BIGINT is recommended to match the data type of RowID. Any specified column constraints, such as NOT NULL or UNIQUE are accepted but ignored.

- Data values are system-generated. They consist of unique, nonzero, positive integers.

- By default, IDENTITY column data values cannot be user-specified. By default, an **INSERT** statement does not, and cannot, specify an IDENTITY column value. Attempting to do so generates an SQLCODE -111 error. To determine whether an IDENTITY column value can be specified, call the **$SYSTEM.SQL.Util.GetOption("IdentityInsert")** method; the default is 0. To change this setting for the current process, call the **$SYSTEM.SQL.Util.SetOption()** method, as follows: SET status=$SYSTEM.SQL.Util.SetOption("IdentityInsert",1,.oldval). You can also specify %CLASSPARAMETER ALLOWIDENTITYINSERT=1 in the table definition. Specifying ALLOWIDENTITYINSERT=1 overrides any setting applied using **SetOption("IdentityInsert")**. For further details, refer to the INSERT statement.

- IDENTITY column data values cannot be modified in an **UPDATE** statement. Attempting to do so generates an SQLCODE -107 error.

- The system automatically projects a primary key on the IDENTITY column to ODBC and JDBC. If a CREATE TABLE or ALTER TABLE statement defines a primary key constraint or a unique constraint on an IDENTITY column, or on a set of columns including an IDENTITY column, the constraint definition is ignored and no corresponding primary key or unique index definition is created.

- A SELECT * statement *does* return a table's IDENTITY column.

Following an **INSERT**, **UPDATE**, or **DELETE** operation, you can use the LAST_IDENTITY function to return the value of the IDENTITY column for the most-recently modified record. If no IDENTITY column is defined, **LAST_IDENTITY** returns the RowID value of the most recently modified record.

These SQL statements create a table with an IDENTITY column and insert a rows into that table, generating an IDENTITY column value for the created table:

SQL

```
CREATE TABLE Employee (
    EmpNum INT NOT NULL,
    MyID IDENTITY NOT NULL,
    Name VARCHAR(30) NOT NULL,
    CONSTRAINT EmployeePK PRIMARY KEY (EmpNum))
```

SQL

```
INSERT INTO Employee (EmpNum,Name)
SELECT ID,Name FROM SQLUser.Person WHERE Age >= '25'
```

In this case, the primary key, EmpNum, is taken from the ID column of another table. EmpNum values are unique integers, but because of the WHERE clause, this column might contain gaps in the sequence. The IDENTITY column, MyID, assigns a user-visible unique sequential integer to each record.

Sharded Table Restrictions

When defining a sharded table, keep these restrictions in mind:

- A sharded table can only be used in a sharded environment; a non-sharded table can be used in a sharded or non-sharded environment. Not all tables are good candidates for sharding. Optimal performance in a sharded environment is generally achieved by using a combination of sharded tables (generally very large tables) and non-sharded tables. For more details, see Evaluating the Benefits of Sharding and Evaluate Existing Tables for Sharding.

- You must define a table as a sharded table either using **CREATE TABLE** or a persistent class definition. You cannot use **ALTER TABLE** to add a shard key to an existing table.

- A UNIQUE column constraint on a sharded table can have a significant negative impact on insert/update performance unless the shard key is a subset of the unique key. For more details, see Evaluate Unique Constraints in Horizontally Scaling InterSystems IRIS for Data Volume with Sharding.

- Sharding a table that is involved in complex transactions requiring atomicity is not recommended.

- A sharded table cannot contain a ROWVERSION data type or SERIAL (%Library.Counter) data type column.

- A sharded table cannot specify the *VERSIONPROPERTY* class parameter.

- To specify a shard key, the current namespace must be configured for sharding. If the current namespace is not configured for sharding, a **CREATE TABLE** that specifies a shard key fails with an SQLCODE -400 error. For details on configuring namespaces for sharding, see Configure the Shard Master Data Server.

- The only referential action supported for sharded tables is NO ACTION. Any other referential action results in an SQLCODE -400 error.

- A shard key column can only take %EXACT, %SQLSTRING, or %SQLUPPER collation, with no truncation. For more details, see Querying the Sharded Cluster.

For more details on sharding, see Create Target Sharded Tables.

Legacy Options

%EXTENTSIZE and %NUMROWS Keywords

The %EXTENTSIZE and %NUMROWS keywords provide an option to store the anticipated number of rows in the table being created. The InterSystems SQL query optimizer uses this value to estimate the cost of query plans. A table can define one or the other of these values but not both. For example:

SQL

```
CREATE TABLE Sample.DaysInAYear (
    %EXTENTSIZE 366,
    MonthName VARCHAR(24),
    Day INTEGER)
```

Starting in 2021.2, the first time you query a table, InterSystems IRIS collects statistics such as the table size automatically. The SQL query optimizer uses these generated statistics to suggest appropriate query plan, making the %EXTENTSIZE and %NUMROWS keywords unnecessary. For more details on optimizing tables with table statisticss, see Table Statistics for Query Optimizer.

%FILE Keyword

The %FILE keyword provides an option to specify a file name that documents the table. For example:

SQL

```
CREATE TABLE Employee (
    %FILE 'C:\SQL\employee_table_desc.txt',
    EmpNum INT PRIMARY KEY,
    NameLast VARCHAR(30) NOT NULL,
    NameFirst VARCHAR(30) NOT NULL,
    StartDate TIMESTAMP %Description 'MM/DD/YY')
```

This keyword is not recommended. Instead, document the table by using the %DESCRIPTION keyword.

Shard Key and %CLASSPARAMETER in Column List Parentheses

Old **CREATE TABLE** code might include the Shard Key definition and %CLASSPARAMETER clauses as comma-separated elements within the table element parentheses. For example: CREATE TABLE myTable(Name VARCHAR(50), DOB DATE, %CLASSPARAMETER USEEXTENTSET = 1). The preferred syntax is to specify these clauses after the closing parenthesis. For example: CREATE TABLE myTable(Name VARCHAR(50), DOB TIMESTAMP) WITH %CLASSPARAMETER USEEXTENTSET = 1. Specifying duplicates of these clauses generates an SQLCODE -327 error.

Options Supported for Compatibility Only

InterSystems SQL accepts the following **CREATE TABLE** options for parsing purposes only, to aid in the conversion of existing SQL code to InterSystems SQL. These options do not provide any actual functionality.

```
{ON | IN} dbspace-name LOCK MODE [ROW | PAGE] [CLUSTERED | NONCLUSTERED] WITH FILLFACTOR = literal
MATCH [FULL | PARTIAL] CHARACTER SET identifier COLLATE identifier  /* But COLLATE keyword is still
used*/ NOT FOR REPLICATION
```

See Also

- ALTER TABLE, DROP TABLE

- SELECT, JOIN

- INSERT, UPDATE, INSERT OR UPDATE

- GRANT

- Defining Tables

- SQL and Object Settings Pages

- SQLCODE error messages

CREATE TRIGGER (SQL)

Creates a trigger.

Synopsis

```
CREATE [OR REPLACE] TRIGGER trigname {BEFORE
| AFTER} event [,event]
[ORDER integer]    ON table    [REFERENCING {OLD | NEW} [ROW] [AS] alias]    action
```

Arguments

Argument	Description
trigname	The name of the trigger to be created, which is an identifier. A trigger name may be qualified or unqualified; if qualified, its schema name must match the table's schema name.
BEFORE *event* AFTER *event*	The time (BEFORE or AFTER) the *event* to execute the trigger. The trigger event, or a comma-separated list of trigger events. Available *event* list options are INSERT, DELETE, and UPDATE. You can specify a single UPDATE OF *event*. The UPDATE OF clause is followed by a column name or a comma-separated list of column names. The UPDATE OF clause can only be specified when LANGUAGE is SQL. The UPDATE OF clause cannot be specified in a comma-separated *event* list.
ORDER *integer*	*Optional* — The order in which triggers should be executed when there are multiple triggers for a table with the same time and event. If order is omitted, a trigger is assigned an order of 0.
ON *table*	The table the trigger is created for. A table name may be qualified or unqualified; if qualified, the trigger must reside in the same schema as the table.
REFERENCING OLD ROW AS *alias* REFERENCING NEW ROW AS *alias*	*Optional* — A REFERENCING clause can only be used when LANGUAGE is SQL. A REFERENCING clause allows you to specify an alias that you can use to reference a column. REFERENCING OLD ROW allows you reference the old value of a column during an UPDATE or DELETE trigger. REFERENCING NEW ROW allows you to reference the new value of a column during an INSERT or UPDATE trigger. The ROW AS keywords are optional. For an UPDATE, you can specify both OLD and NEW in the same REFERENCING clause, as follows: `REFERENCING OLD oldalias NEW newalias`.

Argument	Description
action	The program code for the trigger. The *action* argument can contain various optional keyword clauses, including (in order): a FOR EACH clause; a WHEN clause with a predicate condition governing execution of the triggered action; and a LANGUAGE clause which specifies either LANGUAGE SQL or LANGUAGE OBJECTSCRIPT. If the LANGUAGE clause is omitted, SQL is the default. Following these clauses, you specify one or more lines of either SQL trigger code or ObjectScript trigger code specifying the action to perform when the trigger is executed.

Description

The **CREATE TRIGGER** command defines a trigger, a block of code to be executed when data in a specific table is modified. A trigger is executed ("fired" or "pulled") when a specific triggering event occurs, such as a new row being inserted into a specified table. A trigger executes user-specified trigger code. You can specify that the trigger should execute this code before or after the execution of the triggering event. A trigger is specific to a specified table.

- A trigger is fired by a specified *event*: an **INSERT**, **DELETE**, or **UPDATE** operation. You can specify a comma-separated list of *events* to execute the trigger when any one of the specified events occurs on the specified table.

- A trigger is fired by an *event* either (potentially) multiple times or just once. A row-level trigger is fired once for each row modified. A statement-level trigger is fired once for an *event*. This trigger type is specified using the FOR EACH clause. A row-level trigger is the default trigger type.

- Commonly, firing a trigger code performs an operation on another table or file, such as performing a logging operation or displaying a message. Firing a trigger cannot modify data in the triggering record. For example, if an update to Record 7 fires a trigger, that trigger's code block cannot update or delete Record 7. A trigger can modify the same table that invoked the trigger, but the triggering *event* and the trigger code operation must be different to prevent a recursive trigger infinite loop.

The optional keyword **OR REPLACE** allows you to modify or replace an existing trigger. **CREATE OR REPLACE TRIGGER** has the same effect as invoking **DROP TRIGGER** to delete the old version of the trigger and then invoking **CREATE TRIGGER**. The command **DROP TABLE** drops all triggers associated with that table.

Privileges and Locking

The **CREATE TRIGGER** command is a privileged operation. The user must have %CREATE_TRIGGER administrative privilege to execute **CREATE TRIGGER**. Failing to do so results in an SQLCODE –99 error with the %msg `User 'name' does not have %CREATE_TRIGGER privileges.`

The user must have %ALTER privilege on the specified table. If the user is the Owner (creator) of the table, the user is automatically granted %ALTER privilege for that table. Otherwise, the user must be granted %ALTER privilege for the table. Failing to do so results in an SQLCODE –99 error with the %msg `User 'name' does not have required %ALTER privilege needed to create a trigger on table: 'Schema.TableName'.`

You can use the GRANT command to assign %CREATE_TRIGGER and %ALTER privileges, if you hold appropriate granting privileges.

In embedded SQL, you can use the **$SYSTEM.Security.Login()** method to log in as a user with appropriate privileges:

ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql(     )
```

You must have the **`%Service_Login:Use`** privilege to invoke the **$SYSTEM.Security.Login** method. For further information, refer to %SYSTEM.Security in the *InterSystems Class Reference*.

- **CREATE TRIGGER** cannot be used on a table projected from a persistent class, unless the table class definition includes [DdlAllowed]. Otherwise, the operation fails with an SQLCODE -300 error with the %msg `DDL not enabled for class 'Schema.tablename'`.

- **CREATE TRIGGER** cannot be used on a table projected from a deployed persistent class. This operation fails with an SQLCODE -400 error with the %msg `Unable to execute DDL that modifies a deployed class: 'classname'`.

The **CREATE TRIGGER** statement acquires a table-level lock on *table*. This prevents other processes from modifying the table's data. This lock is automatically released at the conclusion of the **CREATE TRIGGER** operation.

To create a trigger, the table cannot be locked by another process in either EXCLUSIVE MODE or SHARE MODE. Attempting a **CREATE TRIGGER** operation on a locked table results in an SQLCODE -110 error, with a %msg such as the following: `Unable to acquire exclusive table lock for table 'Sample.MyTest'`.

Other Ways of Defining Triggers

You can define an SQL trigger as a class object as described in Trigger Definitions. The following is an example of an Object trigger:

Class Member

```
Trigger SQLJournal [ CodeMode = objectgenerator, Event = INSERT/UPDATE, ForEach = ROW/OBJECT, Time =
AFTER ]
{   /* ObjectScript trigger code
     that updates a journal file
     after a row is inserted or updated. */
}
```

Arguments

trigname

A trigger name follows the same identifier requirements as a table name, but not the same uniqueness requirements. A trigger name should be unique for all tables within a schema. Thus, triggers referencing different tables in a schema should not have the same name. Violating this uniqueness requirement can result in a **DROP TRIGGER** error.

A trigger and its associated table must reside in the same schema. You cannot use the same name for a trigger and a table in the same schema. Violating trigger naming conventions results in an SQLCODE -400 error at **CREATE TRIGGER** execution time.

A trigger name may be unqualified or qualified. A qualified trigger name has the form:

```
schema.trigger
```

If the trigger name is unqualified, the trigger schema name defaults to the same schema as the specified table schema. If the table name is unqualified, the table schema name defaults to the same schema as the specified trigger schema. If both are unqualified, the default schema name is used; schema search paths are not used. If both are qualified, the trigger schema name must be the same as the table schema name. A schema name mismatch results in an SQLCODE -366 error; this should only occur when both the trigger name and the table name are qualified and they specify different schema names.

Trigger names follow identifier conventions, subject to the restrictions below. By default, trigger names are simple identifiers. A trigger name should not exceed 128 characters. Trigger names are not case-sensitive.

InterSystems IRIS uses *trigname* to generate a corresponding trigger name in the InterSystems IRIS class. The corresponding class trigger name contains only alphanumeric characters (letters and numbers) and is a maximum of 96 characters in length. To generate this identifier name, InterSystems IRIS first strips punctuation characters from the trigger name, and then

generates a unique identifier of 96 (or less) characters, substituting a number for the 96th character when needed to create a unique name. This name generation imposes the following restrictions on the naming of triggers:

- A trigger name must include at least one letter. Either the first character of the trigger name or the first character after initial punctuation characters must be a letter.

- InterSystems IRIS supports 16-bit (wide) characters for trigger names. A character is a valid letter if it passes the $ZNAME test.

- Because names generated for an InterSystems IRIS class do not include punctuation characters, it is not advisable (though possible) to create trigger names that differ only in their punctuation characters.

- A trigger name may be much longer than 96 characters, but trigger names that differ in their first 96 alphanumeric characters are much easier to work with.

Issuing a **CREATE TRIGGER** with the name of an existing trigger issues an SQLCODE -365 "Trigger name not unique" error. Use the optional **OR REPLACE** keyword or drop the old trigger first with **DROP TRIGGER**.

If two triggers referencing different tables in a schema have the same name, a **DROP TRIGGER** may issue an SQLCODE -365 "Trigger name not unique" error with the message "Trigger 'MyTrigName' found in 2 classes"

event

The time that the trigger is fired is specified by the BEFORE or AFTER keyword; these keywords specify that the trigger operation should occur either before or after InterSystems IRIS executes the triggering *event*. A BEFORE trigger is executed before performing the specified *event*, but *after* verifying the *event*. For example, InterSystems IRIS only executes a BEFORE DELETE trigger if the **DELETE** statement is valid for the specified row(s), and the process has the necessary privileges to perform the **DELETE**, including any foreign key referential integrity checks. If the process cannot perform the specified *event*, InterSystems IRIS issues an error code for the *event*; it does not execute the BEFORE trigger.

The BEFORE or AFTER keyword is followed by the name of a triggering event, or a comma-separated list of triggering events. A trigger specified as INSERT is executed when a row is inserted into the specified table. A trigger specified as DELETE is executed when a row is deleted from the specified table. A trigger specified as UPDATE is executed when a row is updated in the specified table. You can specify a single trigger event or a comma-separated list of INSERT, UPDATE, or DELETE trigger events in any order.

A trigger specified as UPDATE OF is executed only when one or more of the specified columns is updated in a row in the specified table. Column names are specified as a comma-separated list. Column names can be specified in any order. An UPDATE OF trigger has the following restrictions:

- UPDATE OF is only valid if the trigger code language is SQL (the default); an SQLCODE -50 error is issued if the trigger code language is OBJECTSCRIPT.

- UPDATE OF cannot be combined with other triggering events; an SQLCODE -1 error is issued if you specify UPDATE OF in a comma-separated list of triggering events.

- UPDATE OF cannot specify a non-existent field; an SQLCODE -400 error is issued.

- UPDATE OF cannot specify a duplicate field name; an SQLCODE -58 error is issued.

The following are examples of *event* types:

SQL

```
CREATE TRIGGER TrigBI BEFORE INSERT ON Sample.Person
      INSERT INTO TLog (Text) VALUES ('before insert')
```

SQL

```
CREATE TRIGGER TrigAU AFTER UPDATE ON Sample.Person
      INSERT INTO TLog (Text) VALUES ('after update')
```

SQL

```
CREATE TRIGGER TrigBUOF BEFORE UPDATE OF Home_Street,Home_City,Home_State ON Sample.Person
        INSERT INTO TLog (Text) VALUES ('before address update')

CREATE TRIGGER TrigAD AFTER UPDATE,DELETE ON Sample.Person
        INSERT INTO TLog (Text) VALUES ('after update or delete')
```

ORDER

The ORDER clause determines the order in which triggers are executed when there are multiple triggers for the same table with the same time and event. For example, two AFTER DELETE triggers. The trigger with the lowest ORDER integer is executed first, then the next higher integer, and so on. If the ORDER clause is not specified, a trigger is created with an assigned ORDER number of 0 (zero). Thus, triggers with no ORDER clause are always executed before triggers with ORDER clauses.

You can assign the same order value to multiple triggers. You can also create multiple triggers with an (implicit or explicit) order of 0. Multiple triggers with the same time, event, and order are executed together in random order.

Triggers are executed in the sequence: time > order > event. Thus if you have a BEFORE INSERT trigger and a BEFORE INSERT,UPDATE trigger, the trigger with the lowest ORDER value would be executed first. If you have a BEFORE INSERT trigger and a BEFORE INSERT,UPDATE trigger with the same ORDER value, the INSERT is executed before the INSERT,UPDATE. This is because — time and order being the same — a single-event trigger is always executed before a multi-event trigger. If two (or more) triggers have identical time, order, and event values, the order of execution is random.

The following examples show how ORDER numbers work. All of these **CREATE TRIGGER** statements create triggers that are executed by the same event:

SQL

```
CREATE TRIGGER TrigA BEFORE DELETE ON doctable
        INSERT INTO TLog (Text) VALUES ('doc deleted')
  /* Assigned ORDER=0 */
```

SQL

```
CREATE TRIGGER TrigB BEFORE DELETE ORDER 4 ON doctable
        INSERT INTO TReport (Text) VALUES ('doc deleted')
  /* Specified as ORDER=4 */
```

SQL

```
CREATE TRIGGER TrigC BEFORE DELETE ORDER 2 ON doctable
        INSERT INTO Ttemps (Text) VALUES ('doc deleted')
  /* Specified as ORDER=2 */
```

SQL

```
CREATE TRIGGER TrigD BEFORE DELETE ON doctable
        INSERT INTO Tflags (Text) VALUES ('doc deleted')
  /* Also assigned ORDER=0 */
```

These triggers will execute in the sequence: (TrigA, TrigD), TrigC, TrigB. Note that TrigA and TrigD have the same order number, and thus execute in random sequence.

REFERENCING

The REFERENCING clause can specify an alias for the old value of a row, the new value of a row, or both. The old value is the row value before the triggered action of an UPDATE or DELETE trigger. The new value is the row value after the

triggered action of an UPDATE or INSERT trigger. For an UPDATE trigger, you can specify aliases for both the before and after row values, as follows:

```
REFERENCING OLD ROW AS oldalias NEW ROW AS newalias
```

The keywords ROW and AS are optional. Therefore, the same clause can also be specified as:

```
REFERENCING OLD oldalias NEW newalias
```

It is not meaningful to refer to an OLD value before an INSERT or a NEW value after a DELETE. Attempting to do so results in an SQLCODE -48 error at compile time.

A REFERENCING clause can only be used when the *action* program code is SQL. Specifying a REFERENCING clause with the LANGUAGE OBJECTSCRIPT clause results in an SQLCODE -49 error.

The following is an example of using REFERENCING with an INSERT:

SQL

```
CREATE TRIGGER TrigA AFTER INSERT ON doctable
      REFERENCING NEW ROW AS new_row
BEGIN
      INSERT INTO Log_Table VALUES ('INSERT into doctable');
      INSERT INTO New_Log_Table VALUES ('INSERT into doctable',new_row.ID);
END
```

action

A triggered action consists of the following elements:

- An optional FOR EACH clause. The available values are FOR EACH ROW, FOR EACH ROW_AND_OBJECT, and FOR EACH STATEMENT. The default is FOR EACH ROW:

  - FOR EACH ROW — This trigger is fired by each row affected by the triggering statement. Note that row-level triggers are not supported for TSQL.

  - FOR EACH ROW_AND_OBJECT — This trigger is fired by each row affected by the triggering statement or by changes via object access. Note that row-level triggers are not supported for TSQL.

    This option defines a *unified trigger*, so called because it is fired by data changes that occur via SQL or object access. (In contrast, with other triggers, if you want to use the same logic when changes occur via object access, it is necessary to implement callbacks such as **%OnDelete()**.)

  - FOR EACH STATEMENT — This trigger is fired once for the whole statement. Statement-level triggers are supported for both ObjectScript and TSQL triggers.

  For the corresponding trigger class options, see FOREACH.

  You can list the FOR EACH value for each trigger using the ACTIONORIENTATION property of INFORMATION.SCHEMA.TRIGGERS.

- An optional WHEN clause, consisting of the WHEN keyword followed by a predicate condition (simple or complex) enclosed in parentheses. If the predicate condition evaluates to TRUE, the trigger is executed. A WHEN clause can only be used when LANGUAGE is SQL. The WHEN clause can reference *oldalias* or *newalias* values. For further details on predicate condition expressions and a list of available predicates, refer to the Overview of Predicates page in this document.

- An optional LANGUAGE clause, either LANGUAGE SQL or LANGUAGE OBJECTSCRIPT. The default is LAN-GUAGE SQL.

- User-written code that is executed when the trigger is executed.

SQL Trigger Code

If LANGUAGE SQL (the default), the triggered statement is an SQL procedure block, consisting of either one SQL procedure statement followed by a semicolon, or the keyword BEGIN followed by one or more SQL procedure statements, each followed by a semicolon, concluding with an END keyword.

A triggered action is atomic, it is either fully applied or not at all, and cannot contain **COMMIT** or **ROLLBACK** statements. The keyword BEGIN ATOMIC is synonymous with the keyword BEGIN.

If LANGUAGE SQL, the **CREATE TRIGGER** statement can optionally contain a REFERENCING clause, a WHEN clause, and/or an UPDATE OF clause. An UPDATE OF clause specifies that the trigger should only be executed when an **UPDATE** is performed on one or more of the columns specified for this trigger. A **CREATE TRIGGER** statement with LANGUAGE OBJECTSCRIPT cannot contain these clauses.

SQL trigger code is executed as embedded SQL. This means that InterSystems IRIS converts SQL trigger code to ObjectScript; therefore, if you view the class definition corresponding to your SQL trigger code, you will see `Language=objectscript` in the trigger definition.

When executing SQL trigger code, the system automatically resets (NEWs) all variable used in the trigger code. After the execution of each SQL statement, InterSystems IRIS checks SQLCODE. If an error occurs, InterSystems IRIS sets the %ok variable to 0, aborting and rolling back both the trigger code operation(s) and the associated **INSERT**, **UPDATE**, or **DELETE**.

ObjectScript Trigger Code

If LANGUAGE OBJECTSCRIPT, the **CREATE TRIGGER** statement cannot contain a REFERENCING clause, a WHEN clause, or an UPDATE OF clause. Specifying these SQL-only clauses with LANGUAGE OBJECTSCRIPT results in compile-time SQLCODE errors -49, -57, or -50, respectively.

If LANGUAGE OBJECTSCRIPT, the triggered statement is a block of one or more ObjectScript statements, enclosed by curly braces.

Because the code for a trigger is not generated as a procedure, all local variables in a trigger are public variables. This means all variables in triggers should be explicitly declared with a **NEW** statement; this protects them from conflicting with variables in the code that invokes the trigger.

If trigger code contains Macro Preprocessor statements (# commands, ## functions, or $$$macro references), these statements are compiled *before* the **CREATE TRIGGER** DDL code itself.

ObjectScript trigger code can contain Embedded SQL.

You can issue an error from trigger code by setting the %ok variable to 0. This creates a runtime error that aborts and rolls back execution of the trigger. It generates the appropriate SQLCODE error (for example, SQLCODE -131 "After insert trigger failed") and returns the user-specified value of the %msg variable as a string to describe the cause of the trigger code error. Note that setting %ok to a non-numeric value sets %ok=0.

The system generates trigger code only once, even for a multiple-event trigger.

Field References and Pseudo-field References

Trigger code written in ObjectScript can contain field references, specified as {*fieldname*}, where *fieldname* specifies an existing field in the current table. No blank spaces are permitted within the curly braces.

You can follow the *fieldname* with *N (new), *O (old), or *C (compare) to specify how to handle an inserted, updated, or deleted field data value, as follows:

- {*fieldname**N}

    - For **UPDATE**, returns the new field value after the specified change is made.

    - For **INSERT**, returns the value inserted.

- For **DELETE**, returns the value of the field before the delete.

- {*fieldname**O}

  - For **UPDATE**, returns the old field value before the specified change is made.

  - For **INSERT**, returns NULL.

  - For **DELETE**, returns the value of the field before the delete.

- {*fieldname**C}

  - For **UPDATE**, returns 1 (TRUE) if the new value differs from the old value, otherwise returns 0 (FALSE).

  - For **INSERT**, returns 1 (TRUE) if the inserted value is non-NULL, otherwise returns 0 (FALSE).

  - For **DELETE**, returns 1 (TRUE) if the value being deleted is non-NULL, otherwise returns 0 (FALSE).

For **UPDATE**, **INSERT**, or **DELETE**, {*fieldname*} returns the same value as {*fieldname**N}.

For example, the following trigger returns the Name field value for a new row inserted into Sample.Employee. (You can perform the **INSERT** from the SQL Shell to view this result):

```
CREATE TRIGGER InsertNameTrig AFTER INSERT ON Sample.Employee
   LANGUAGE OBJECTSCRIPT
   {WRITE "The employee ",{Name*N}," was ",{%%OPERATION},"ed on ",{%%TABLENAME},!}
```

Line returns are not permitted within a statement that sets a field value. For further details, refer to the SqlComputeCode property keyword in the *Class Definition Reference*.

You can use the **GetAllColumns()** method to list the field names defined for a table. For further details, refer to Column Names and Numbers.

Trigger code written in ObjectScript can also contain the pseudo-field reference variables {%%CLASSNAME}, {%%CLASSNAMEQ}, {%%OPERATION}, {%%TABLENAME}, and {%%ID}. The pseudo-fields are translated into a specific value at class compilation time. All of these pseudo-field keywords are not case-sensitive.

- {%%CLASSNAME} and {%%CLASSNAMEQ} both translate to the name of the class which projected the SQL table definition. {%%CLASSNAME} returns an unquoted string and {%%CLASSNAMEQ} returns a quoted string.

- {%%OPERATION} translates to a string literal, either INSERT, UPDATE, or DELETE, depending on the operation that invoked the trigger.

- {%%TABLENAME} translates to the fully qualified name of the table.

- {%%ID} translates to the RowID name. This reference is useful when you do not know the name of the RowID field.

Referencing Stream Property

When a Stream field/property is referenced in a trigger definition, like {StreamField}, {StreamField*O}, or {StreamField*N}, the value of the {StreamField} reference is the stream's OID (object ID) value.

For a BEFORE INSERT or BEFORE UPDATE trigger, if a new value is specified by the INSERT/UPDATE/ObjectSave, the {StreamField*N} value will be either the OID of the temporary stream object, or the new literal stream value. For a BEFORE UPDATE trigger, if a new value is not specified for the stream field/property, {StreamField*O} and {Stream-Field*N} will both be the OID of the current field/property stream object.

Referencing SQLComputed Property

When a transient SqlComputed field/property (either "Calculated" or explicitly "Transient") is referenced in a trigger defi-nition, Get()/Set() method overrides are not recognized by the trigger. Use SQLCOMPUTED/SQLCOMPUTONCHANGE, rather than overriding the property's Get() or Set() method.

Using Get()/Set() method overrides can result in the following erroneous result: The {property*O} value is determined using SQL and does not use the overridden Get()/Set() methods. Because the property is not stored on disk, {property*O} uses the SqlComputeCode to "recreate" the old value. However, {property*N} uses the overridden Get()/Set() methods to access the property's value. As a result, there is a possibility for {property*O} and {property*N} to be different (and thus {property*C}=1) even though the property did not actually change.

Labels

Trigger code may contain line labels (tags). To specify a label in trigger code, prefix the label line with a colon to indicate that this line should begin in the first column. InterSystems IRIS strips out the colon and treats the remaining line as a label. However, because trigger code is generated outside the scope of any procedure blocks, every label must be unique throughout the class definition. Any other code compiled into the class's routine must not have the same label defined, including in other triggers, in non-procedure block methods, SqlComputeCode, and other code.

Note: This use of a colon prefix for a label takes precedence over the use of a colon prefix for a host variable reference. To avoid this conflict, it is recommended that embedded SQL trigger code lines never begin with a host variable reference. If you must begin a trigger code line with a host variable reference, you can designate it as a host variable (and not a label) by doubling the colon prefix.

Method Calls

You can call class methods from within trigger code, because class methods do not depend on having an open object. You must use the ##class(classname).Method() syntax to invoke a method. You cannot use the ..Method() syntax, because this syntax requires a current open object.

You can pass the value of a field of the current row as an argument of the class method, but the class method itself cannot use field syntax.

Listing Existing Triggers

You can use the INFORMATION.SCHEMA.TRIGGERS class to list the currently defined triggers. This class lists for each trigger the name of the trigger, the associated schema and table name, and the trigger creation timestamp. For each trigger it lists the EVENT_MANIPULATION property (INSERT, UPDATE, DELETE, INSERT/UPDATE, INSERT/UPDATE/DELETE) and ACTION_TIMING property (BEFORE, AFTER). It also lists the ACTION_STATEMENT, which is the generated SQL trigger code.

Trigger Runtime Errors

A trigger and its invoking event execute as an atomic operation on a single row basis. That is:

- A failed BEFORE trigger is rolled back, the associated **INSERT**, **UPDATE**, or **DELETE** operation is not executed, and all locks on the row are released.

- A failed AFTER trigger is rolled back, the associated **INSERT**, **UPDATE**, or **DELETE** operation is rolled back, and all locks on the row are released.

- A failed **INSERT**, **UPDATE**, or **DELETE** operation is rolled back, the associated BEFORE trigger is rolled back, and all locks on the row are released.

- A failed **INSERT**, **UPDATE**, or **DELETE** operation is rolled back, the associated AFTER trigger is not executed, and all locks on the row are released.

Note that integrity is maintained for the current row operation only. Your application program must handle data integrity issues involving operation on multiple rows by using transaction processing statements.

Because a trigger is an atomic operation, you cannot code transaction statements, such as commits and rollbacks, within trigger code.

If an **INSERT**, **UPDATE**, or **DELETE** operation causes multiple triggers to execute, the failure of one trigger causes all remaining triggers to remain unexecuted.

- SQLCODE -415: If there is an error in the trigger code (for example, a reference to a non-existent table or an undefined variable) execution of the trigger code fails at runtime and InterSystems IRIS issues an SQLCODE -415 error "Fatal error occurred within the SQL filer".

- SQLCODE -130 through -135: When a trigger operation fails, InterSystems IRIS issues one of the SQLCODE error codes -130 through -135 at runtime indicating the type of trigger that failed. You can force a trigger to fail by setting the %ok variable to 0 in the trigger code. This issues the appropriate SQLCODE error (for example, SQLCODE -131 "After insert trigger failed") and returns the user-specified value of the %msg variable as a string to describe the cause of the trigger code error.

Examples

The following example demonstrates **CREATE TRIGGER** with an ObjectScript DELETE trigger. It assumes that there is a data table (TestDummy) that contains records. It creates a log table (TestDummyLog) and a DELETE trigger that writes to the log table when a delete is performed on the data table. The trigger inserts the name of the data table, the RowId of the deleted row, the current date, and the type of operation performed (the **%oper** special variable), in this case "DELETE":

SQL

```
CREATE TABLE TestDummyLog
    (TableName VARCHAR(40),
     IDVal INTEGER,
     LogDate DATE,
     Operation VARCHAR(40))
```

ObjectScript

```
&sql(CREATE TRIGGER TrigTestDummy AFTER DELETE ON TestDummy
    LANGUAGE OBJECTSCRIPT {
    NEW id
    SET id = {ID}
    &sql(INSERT INTO TestDummyLog (TableName,IDVal,LogDate,Operation)
        VALUES ('TestDummy',:id,+$HOROLOG,:%oper))
    }
 )
 WRITE !,"SQL trigger code is: ",SQLCODE
```

The following examples demonstrate CREATE TRIGGER with an SQL INSERT trigger. The first program creates a table, an INSERT trigger for that table, and a log table for the trigger's use. The second program issues an INSERT against the table, which invokes the trigger, which logs an entry in the log table. After displaying the log entry, the program drops both tables so that this program can be run repeatedly:

SQL

```
  CREATE TABLE TestDummy (
     testnum     INT NOT NULL,
     firstword   CHAR (30) NOT NULL,
     lastword    CHAR (30) NOT NULL,
     CONSTRAINT TestDummyPK PRIMARY KEY (testnum))
CREATE TABLE TestDummyLog (
     entry CHAR (60) NOT NULL)
  )
CREATE TRIGGER TrigTestDummy AFTER INSERT ON TestDummy
   BEGIN
     INSERT INTO TestDummyLog (entry) VALUES
     (CURRENT_TIMESTAMP||' INSERT to TestDummy');
   END
```

SQL

```
  INSERT INTO TestDummy (testnum,firstword,lastword) VALUES
    (46639,'hello','goodbye'))
SELECT entry FROM TestDummyLog
DROP TABLE TestDummy
DROP TABLE TestDummyLog
```

The following example includes a WHEN clause that specifies that the *action* should only be performed when the predicate condition in parentheses is met:

SQL

```
CREATE TRIGGER Trigger_2 AFTER INSERT ON Table_1
  WHEN (f1 %STARTSWITH 'A')
  BEGIN
    INSERT INTO Log_Table VALUES (new_row.Category);
  END
```

The following example defines a trigger that returns the old Name field value and the new Name field value after a row is inserted, updated, or deleted in Sample.Employee. (You can perform the triggering event operation from the SQL Shell to view this result):

```
CREATE TRIGGER EmployNameTrig AFTER INSERT,UPDATE,DELETE ON Sample.Employee
  LANGUAGE OBJECTSCRIPT
  {WRITE "Employee old name:",{Name*O}," new name:",{Name*N}," ",{%%OPERATION}," on ",{%%TABLENAME},!}
```

See Also

- DROP TRIGGER

- GRANT

- Using Triggers

- SQLCODE error messages

CREATE USER (SQL)

Creates a user account.

Synopsis

```
CREATE USER user-name IDENTIFY BY password
```

Description

The **CREATE USER** command creates a user account with the specified password.

A *user-name* can be any valid identifier of up to 160 characters. A *user-name* must follow identifier naming conventions. A *user-name* can contain Unicode characters. User names are not case-sensitive.

A *user-name* specified as a delimited identifier can be an SQL reserved word and can contain a comma (,), period (.), caret (^), and the two-character arrow sequence (->). It may begin with any valid character except the asterisk (*).

The IDENTIFY BY and IDENTIFIED BY keywords are synonyms.

A *password* can be a numeric literal, an identifier, or a quoted string. A numeric literal or an identifier does not have to be enclosed in quotes. A quoted string is commonly used to include blanks in a password; a quoted password can contain any combination of characters, with the exception of the quote character itself. A numeric literal must consist of only the characters 0 through 9. An identifier must start with a letter (uppercase or lowercase) or a % (percent symbol); this can be followed by any combination of letters, numbers, or any of the following symbols: _ (underscore), & (ampersand), $ (dollar sign), or @ (at sign).

Passwords are case-sensitive. A password must be at least three characters, and less than 33 characters, in length. Specifying a password that is too long or too short generates an SQLCODE -400 error, with a %msg value of "ERROR #845: Password does not match length or pattern requirements".

You cannot use a host variable to specify a *user-name* or *password* value.

Creating a user does not create any roles or grant any roles to the user. Instead, the user is given permissions for the database they are logging in to, and USE permission on the %SQL/Service service if the user holds at least one SQL privilege in the namespace. To assign privileges or roles to a user, use the **GRANT** command. To create roles, use the **CREATE ROLE** command.

If you invoke **CREATE USER** to create a user that already exists, SQL issues an SQLCODE -118 error, with a %msg value of "User named '*name*' already exists". You can determine if a user already exists by invoking the **$SYSTEM.SQL.Security.UserExists()** method:

ObjectScript

```
WRITE $SYSTEM.SQL.Security.UserExists("Admin"),!
WRITE $SYSTEM.SQL.Security.UserExists("BertieWooster")
```

This method returns 1 if the specified user exists, and 0 if the user does not exist. User names are not case-sensitive.

Privileges

The **CREATE USER** command is a privileged operation. Prior to using **CREATE USER** in embedded SQL, you must be logged in as a user with one of the following:

- The %Admin_Secure administrative resource with USE permission

- The %Admin_UserEdit administrative resource with USE permission

- Full security privileges on the system

If you are not, the **CREATE USER** command results in an SQLCODE -99 error (Privilege Violation).

Use the **$SYSTEM.Security.Login()** method to assign a user with appropriate privileges:

ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql( /* SQL code here */  )
```

You must have the `%Service_Login:Use` privilege to invoke the **$SYSTEM.Security.Login** method. For further information, refer to %SYSTEM.Security in the *InterSystems Class Reference*.

Arguments

user-name

The name of the user to be created. The name is an identifier with a maximum of 128 characters. It can contain Unicode letters. *user-name* is not case-sensitive.

password

The password for this user. A *password* must be at least 3 characters, and cannot exceed 32 characters. Passwords are case-sensitive. Passwords can contain Unicode characters.

Example

The following embedded SQL example creates a new user named "BillTest" with a password of "Carl4SHK". (The **$RANDOM** toggle is provided so that you can execute this example program repeatedly.)

ObjectScript

```
Main
    DO $SYSTEM.Security.Login("_SYSTEM","SYS")
    SET x=$SYSTEM.SQL.Security.UserExists("BillTest")
    IF x=0 {&sql(CREATE USER BillTest IDENTIFY BY Carl4SHK)
            IF SQLCODE '= 0 {WRITE "CREATE USER error: ",SQLCODE,!
                             QUIT}
        }
    WRITE "User BillTest exists",!
Cleanup
    SET toggle=$RANDOM(2)
    IF toggle=0 {
      &sql(DROP USER BillTest)
      IF SQLCODE '= 0 {WRITE "DROP USER error: ",SQLCODE,!}
      }
    ELSE {WRITE !,"No drop this time",!}
    WRITE "User BillTest exists? ",$SYSTEM.SQL.Security.UserExists("BillTest"),!
    QUIT
```

See Also

- SQL statements: ALTER USER, DROP USER, GRANT, REVOKE, CREATE ROLE

- SQL Users, Roles, and Privileges

- SQLCODE error messages

- ObjectScript: $ROLES and $USERNAME special variables

CREATE VIEW (SQL)

Creates a view.

Synopsis

```
CREATE [OR REPLACE] VIEW view-name [(column-commalist)]
    AS select-statement
    [ WITH READ ONLY  |  WITH [level] CHECK OPTION ]
```

Description

The **CREATE VIEW** command defines the content of a view. The **SELECT** statement that defines the view can reference more than one table and can reference other views.

Privileges

The **CREATE VIEW** command is a privileged operation. The user must have %CREATE_VIEW administrative privilege to execute **CREATE VIEW**. Failing to do so results in an SQLCODE –99 error with the %msg User 'name' does not have %CREATE_VIEW privileges. You can use the GRANT command to assign %CREATE_VIEW privileges, if you hold appropriate granting privileges.

To select from the objects referenced in the **SELECT** clause of a view being created, it is necessary to have the appropriate privileges:

- When creating a view using Dynamic SQL or via a database driver, you must have SELECT privileges on all the columns selected from the underlying tables (or views) referenced by the view. If you do not have SELECT privilege for a specified table (or view) the **CREATE VIEW** command will not execute.

  However, when compiling a class that projects a defined view, these SELECT privileges are not enforced on the columns selected from the underlying tables (or views) referenced by the view. For example, if you create a view using a privileged routine (that has these SELECT privileges), you can later compile the view class, because you are the owner of the view, regardless of whether you have SELECT privileges for the tables referenced by the view.

- To receive SELECT privilege WITH GRANT OPTION for a view, you must have WITH GRANT OPTION for every table (or view) referenced by the view.

- To receive INSERT, UPDATE, DELETE, or REFERENCES privilege for a view, you must have the same privilege for every table (or view) referenced by the view. To receive WITH GRANT OPTION for any of these privileges, you must hold the privilege WITH GRANT OPTION on the underlying tables.

- If the view is specified WITH READ ONLY, the view is not granted INSERT, UPDATE, or DELETE privileges, regardless of the privileges you hold for the underlying tables. If the view is later redefined as read/write, these privileges are added when the class projecting the view is recompiled.

You can determine if the current user has these table-level privileges by invoking the %CHECKPRIV command. You can determine if a specified user has these table-level privileges by invoking the **$SYSTEM.SQL.Security.CheckPrivilege()** method. For privilege assignment, refer to the GRANT command.

The creator (owner) of a view is granted the %ALTER privilege WITH GRANT OPTION when the view is compiled.

In embedded SQL, you can use the **$SYSTEM.Security.Login()** method to log in as a user with appropriate privileges:

ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql(       )
```

You must have the **`%Service_Login:Use`** privilege to invoke the **$SYSTEM.Security.Login** method. For further information, refer to %SYSTEM.Security in the *InterSystems Class Reference*.

%CREATE_VIEW privileges are assigned using the GRANT command, which requires you to assign this privilege to a user or role. By default, **CREATE VIEW** security privileges are enforced. This privileges requirement is configurable system-wide using the **$SYSTEM.SQL.Util.SetOption()** method SET `status=$SYSTEM.SQL.Util.SetOption("SQLSecurity",0,.oldval);` to determine the current setting, call the **$SYSTEM.SQL.CurrentSettings()** method, which displays an `SQL security enabled` setting.

The default is 1 (enabled). When SQL Security is enabled, a user can only perform actions on a table or view for which that user has been granted privilege. This is the recommended setting for this option.

If this method is set to 0, SQL Security is disabled for any new process started after changing this setting. This means privilege-based table/view security is suppressed. You can create a table without specifying a user. In this case, Dynamic SQL assigns "_SYSTEM" as user, and Embedded SQL assigns "" (the empty string) as user. Any user can perform actions on a table or view even if that user has no privileges to do so.

View Naming Conventions

A view name has the same naming conventions as a table name, and shares the same name set. Therefore, you cannot use the same name for a table and a view in the same schema. Attempting to do so results in an SQLCODE -201 error. To determine if a table already exists in the current namespace, use the **$SYSTEM.SQL.Schema.TableExists("schema.tname")** method. A class that projects a table definition and a view definition with the same name also generates an SQLCODE -201 error.

View names follow identifier conventions, subject to the restrictions below. By default, view names are simple identifiers. A view name should not exceed 128 characters. View names are not case-sensitive.

InterSystems IRIS uses the view name to generate a corresponding class name. A class name contains only alphanumeric characters (letters and numbers) and must be unique within the first 96 characters. To generate this class name, InterSystems IRIS first strips punctuation characters from the view name, and then generates a identifier that is unique within the first 96 characters, substituting an integer (beginning with 0) for the final character when needed to create a unique class name. InterSystems IRIS generates a unique class name from a valid view name, but this name generation imposes the following restrictions on the naming of views:

- A view name must include at least one letter. Either the first character of the view name or the first character after initial punctuation characters must be a letter.

- InterSystems IRIS supports 16-bit (wide) characters for view names. A character is a valid letter if it passes the $ZNAME test.

- If the first character of the view name is a punctuation character, the second character cannot be a number. This results in an SQLCODE -400 error, with a %msg value of "ERROR #5053: Class name 'schema.name' is invalid" (without the punctuation character). For example, specifying the view name %7A generates the %msg "ERROR #5053: Class name 'User.7A' is invalid".

- Because generated class names do not include punctuation characters, it is not advisable (though possible) to create a view name that differs from an existing view or table name only in its punctuation characters. In this case, InterSystems IRIS substitutes an integer (beginning with 0) for the final character of the name to create a unique class name.

- A view name may be much longer than 96 characters, but view names that differ in their first 96 alphanumeric characters are much easier to work with.

A view name can be qualified or unqualified.

A qualified view name (`schema.viewname`) can specify an existing schema or a new schema. If it specifies a new schema, the system creates that schema.

An unqualified view name (`viewname`) takes the default schema name.

Existing View

To determine if a specified view already exists in the current namespace, use the **$SYSTEM.SQL.Schema.ViewExists("schema.vname")** method.

What happens when you try to create a view that has the same name as an existing view depends on the optional OR REPLACE keyword and on the configuration setting.

With OR REPLACE

If you specify **CREATE OR REPLACE VIEW**, the existing view is replaced by the view definition specified in the **SELECT** clause and any specified WITH READ ONLY or WITH CHECK OPTION. This is the same as performing the corresponding ALTER VIEW statement. Any privileges that had been granted to the original view remain. Ownership of the view transfers to the user who executes the **CREATE OR REPLACE VIEW** statement.

This keyword phrase provides no functionality not available through **ALTER VIEW**. It is provided for compatibility with Oracle SQL code.

Without OR REPLACE

By default, if you specify **CREATE VIEW**, InterSystems IRIS rejects an attempt to create a view with the name of an existing view and issues an SQLCODE -201 error. To determine the current setting, call **$SYSTEM.SQL.CurrentSettings()**, which displays a `Allow DDL CREATE TABLE or CREATE VIEW for existing table or view` setting. The default is 0 (No), which is the recommended setting. If this option is set to 1 (Yes), InterSystems IRIS deletes the class definition associated with the view and then recreates it. This is the same as performing a **DROP VIEW** and then performing a **CREATE VIEW**. Note that this setting affects both **CREATE VIEW** and **CREATE TABLE**.

From the Management Portal, **System Administration**, **Configuration**, **SQL and Object Settings**, **SQL** you can set this option (and other similar create, alter, and drop options) system-wide by selecting the **Ignore redundant DDL statements** check box.

Column Names

A view can optionally include a *column-commalist* list of column names, enclosed in parentheses. These column names, if specified, are the names used to access and display the data for the columns when using that view.

If you omit the *column-commalist*, the following apply:

- The column names of the SELECT source table are used to access and display the data when using the view.

- If any of the SELECT source table column names have column aliases, the column aliases are the names used to access and display the data when using the view.

- If the SELECT source table column names have table aliases, the table aliases are not used in the names used to access and display the data when using the view.

If you omit the list of column names, you must also omit the parentheses.

If you specify the *column-commalist*, the following apply:

- A column name list must specify the enclosing parentheses, even when specifying a single field. You must separate multiple column names with commas. Whitespace and comments are permitted within a *column-commalist*.

- The number of column names must correspond to the number of columns specified in the SELECT statement. Mismatch between the number of view columns and query columns results in an SQLCODE -142 error at compile time.

- The names of column names must be valid identifiers. They may be different names than the SELECT column names, the same names as the SELECT column names, or a combination of both. The specified order of the view column names corresponds to the order of the SELECT column names. Because it is possible to assign a view column the name of an unrelated SELECT column, you must exercise caution when assigning view column names.

- A column name must be unique. Specifying a duplicate column name results in an SQLCODE -97 error. Column names are converted to corresponding class property names by stripping out punctuation characters; column names that differ only in punctuation characters are permitted, but discouraged.

The following example shows a **CREATE VIEW** with matching lists of view columns and query columns:

SQL

```
CREATE VIEW MyView (ViewCol1, ViewCol2, ViewCol3) AS
    SELECT TableCol1, TableCol2, TableCol3
    FROM MyTable
```

Alternatively, you can use the AS keyword in the query to specify the view columns as query column / view column pairs, as shown in the following example:

SQL

```
CREATE VIEW MyView AS
  SELECT TableCol1 AS ViewCol1,
    TableCol2 AS ViewCol2,
    TableCol3 AS ViewCol3
    FROM MyTable
```

SELECT Columns and View Columns

- Data from multiple SELECT columns can be concatenated into a single view column. For example:

  SQL

  ```
  CREATE VIEW MyView (fullname) AS SELECT firstname||' '||lastname FROM MyTable
  ```

- Multiple view columns can refer to the same SELECT column. For example:

  SQL

  ```
  CREATE VIEW MyView (lname,surname) AS SELECT lastname,lastname FROM MyTable
  ```

SELECT Clause Considerations

A view does not have to be a simple subset of the rows and columns of one particular table. A view can be created using a **SELECT** clause of any complexity, specifying any combination of tables or views. There are, however, a few restrictions on the **SELECT** clause of a view definition:

- Can only include an ORDER BY clause if this clause is paired with a TOP clause. If you wish to include all of the rows in the view, you can use a TOP ALL clause. You can include a TOP clause without an ORDER BY clause. However, if you include an ORDER BY clause without a TOP clause, an SQLCODE -143 error is generated. If you project an SQL view from a view class, the query of which contains an ORDER BY clause, the ORDER BY clause is ignored in the view projection.

- Cannot contain host variables. If you attempt to reference a host variable in the **SELECT** clause, the system generates an SQLCODE -148 error.

- Cannot include the INTO keyword. A view that specifies a **SELECT** with an **INTO** clause can be created, but execution of this view fails with an SQLCODE -25 error.

CREATE VIEW can contain a UNION statement to select columns from the union of two tables. You can specify a **UNION** as shown in the following example:

SQL

```
CREATE VIEW MyView (vname,vstate) AS
  SELECT t1.name,t1.home_state
    FROM Sample.Person AS t1
  UNION
  SELECT t2.name,t2.office_state
    FROM Sample.Employee AS t2
```

Note that an unqualified view name, such as in the above example, defaults to the default schema name (for example, the initial schema default SQLUser.MyView), even though the tables referenced by the view are in the Sample schema. Thus it is usually a good practice to always qualify a view name to ensure that it is stored with its associated table(s).

View ID: %vid

When data is accessed through a view, InterSystems IRIS assigns a sequential integer view ID (%vid) to each row returned by that view. Like table row ID numbers, these view row ID numbers are system-assigned, unique, non-zero, non-null, and non-modifiable. This %vid is usually invisible. Unlike a table row ID, it is not displayed when using asterisk syntax; it is only displayed when explicitly specified in the **SELECT**. The %vid can be used to further restrict the number of rows returned by a **SELECT** accessing a view. For further details on using %vid, refer to Defining and Using Views.

Arguments

view-name

The name for the view being created. A valid identifier, subject to the same additional naming restrictions as a table name. A view name can be qualified (schema.viewname), or unqualified (viewname). An unqualified view name takes the default schema name. Note that you cannot use the same name for a table and a view in the same schema.

column-commalist

An optional argument. The column names that compose the view, one or more valid identifiers. If specified, this list is enclosed in parentheses and items in the list are separated by commas.

AS select-statement

A SELECT statement that defines the view.

WITH READ ONLY

An optional argument specifying that no insert, update, or delete operations can be performed through this view upon the table on which the view is based. The default is to permit these operations through a view, subject to the constraints described below.

WITH level CHECK OPTION

An optional argument that specifies how insert, update, or delete operations are performed through this view upon the table on which the view is based. The *level* can be the keywords LOCAL or CASCADED. If no *level* is specified, the WITH CHECK OPTION default is CASCADED.

Updating Through Views

A view can be used to update the tables on which the view is based. You can INSERT new rows through the view, UPDATE data in rows seen through the view, and DELETE rows seen through the view. **INSERT**, **UPDATE**, and **DELETE** statements can be issued for a view, if the **CREATE VIEW** statement specified this ability. To allow updating through a view, specify WITH CHECK OPTION (the default) when defining the view.

Note: If the view is based on a sharded table, you cannot **INSERT**, **UPDATE**, or **DELETE** through a view WITH CHECK OPTION. Attempting to do so results in an SQLCODE -35 with the %msg `INSERT/UPDATE/DELETE not allowed for view (sample.myview) based on sharded table with check option conditions`.

To prevent updating through a view, specify WITH READ ONLY. Attempting an **INSERT**, **UPDATE**, or **DELETE** through a view created WITH READ ONLY generates an SQLCODE -35 error.

In order to update through a view, you must have the appropriate privileges for the table or view to be updated, as specified by the GRANT command.

Updating through views is subject to the following restrictions:

- The view cannot be a class query projected as a view.

- The view's class cannot contain the class parameter READONLY=1.

- The view's **SELECT** statement cannot contain a DISTINCT, TOP, GROUP BY, or HAVING clause, or be part of a UNION.

- The view's **SELECT** statement cannot contain a subquery.

- The view's **SELECT** statement can only list value expressions that are column references.

- The view's **SELECT** statement can have only one table reference; it cannot contain FROM clause JOIN syntax or arrow syntax in the *select-list* or WHERE clause. The table reference must specify either an updateable table or an updateable view.

The WITH CHECK OPTION clause causes an insert or update operation to validate the resulting row against the WHERE clause of the view definition. This ensures that the inserted or modified row is part of the derived view table. There are two available check options:

- WITH LOCAL CHECK OPTION — only the **WHERE** clause of the view specified in the **INSERT** or **UPDATE** statement is checked.

- WITH CASCADED CHECK OPTION — the **WHERE** clause of the view specified in the **INSERT** or **UPDATE** statement and all underlying views are checked. This overrides any WITH LOCAL CHECK OPTION clauses in these underlying views. WITH CASCADED CHECK OPTION is recommended for all updateable views.

  If you specify WITH CHECK OPTION, the check option defaults to CASCADED. The keyword CASCADE is a synonym for CASCADED.

If an **INSERT** operation fails WITH CHECK OPTION validation (as defined above), InterSystems IRIS issues an SQLCODE -136 error.

If an **UPDATE** operation fails WITH CHECK OPTION validation (as defined above), InterSystems IRIS issues an SQL-CODE -137 error.

Examples

The following example creates a view named "CityPhoneBook" from the PhoneBook table:

SQL

```
CREATE VIEW CityPhoneBook AS
    SELECT Name FROM PhoneBook WHERE City='Boston'
```

The following example creates a view named "GuideHistory" from the Guides table. It lists all titles (from the Title column) and whether or not the person is retired:

SQL

```
CREATE VIEW GuideHistory AS
     SELECT Guides, Title, Retired, Date_Retired
     FROM Guides
```

The following example creates the table MyTest, and then creates a view for this table, MyTestView, which selects one field from MyTest:

SQL

```
CREATE TABLE Sample.MyTest (
     TestNum     INT NOT NULL,
     FirstWord   CHAR (30) NOT NULL,
     LastWord    CHAR (30) NOT NULL,
     CONSTRAINT MyTestPK PRIMARY KEY (TestNum))

CREATE VIEW Sample.MyTestView AS
     SELECT FirstWord FROM Sample.MyTest
     WITH CASCADED CHECK OPTION
```

The following example creates a view MyTestView, which selects two fields from MyTest. The SELECT query for this view contains a TOP clause and an ORDER BY clause:

SQL

```
CREATE TABLE Sample.MyTest (
     TestNum     INT NOT NULL,
     FirstWord   CHAR (30) NOT NULL,
     LastWord    CHAR (30) NOT NULL,
     CONSTRAINT MyTestPK PRIMARY KEY (TestNum))

CREATE VIEW Sample.MyTestView AS
     SELECT TOP ALL FirstWord,LastWord FROM Sample.MyTest
     ORDER BY LastWord)
```

The following example creates a view named "StaffWorksDesign" from three tables (Proj, Staff, and Works). The columns Name, Cost, and Project provide the data.

SQL

```
CREATE VIEW StaffWorksDesign (Name,Cost,Project)
     AS SELECT EmpName,Hours*2*Grade,PName
     FROM Proj,Staff,Works
     WHERE Staff.EmpNum=Works.EmpNum
         AND Works.PNum=Proj.PNum AND PType='Design'
```

The following example creates a view named "v_3" by selecting from b.table2 and a.table1 using a **UNION**:

SQL

```
CREATE VIEW v_3(fvarchar)
     AS SELECT DISTINCT *
     FROM
        (SELECT fVARCHAR2 FROM b.table2
         UNION ALL
         SELECT fVARCHAR1 FROM a.table1)
```

See Also

- ALTER VIEW

- DROP VIEW

- CREATE TABLE

- GRANT

- SELECT

- [Defining and Using Views](#)
- [SQL and Object Settings Pages](#)
- [SQLCODE error messages](#)

DECLARE (SQL)

Declares a cursor.

Synopsis

```
DECLARE cursor-name CURSOR FOR query
```

Description

A **DECLARE** statement declares a cursor used in cursor-based Embedded SQL. After declaring a cursor, you issue an OPEN statement to open the cursor and then a series of FETCH statements to retrieve individual records. The cursor defines the **SELECT** query that is used to select records for retrieval by these **FETCH** statements. You issue a CLOSE statement to close (but not delete) the cursor.

As an SQL statement, **DECLARE** is only supported from Embedded SQL. For Dynamic SQL, use instead either a simple **SELECT** statement (with no **INTO** clause), or a combination of Dynamic SQL and Embedded SQL. Equivalent operations are supported through ODBC using the ODBC API.

DECLARE declares a forward-only (non-scrollable) cursor. Fetch operations begin with the first record in the query result set and proceed sequentially through the result set records. A **FETCH** can only fetch a record once. The next **FETCH** fetches the next sequential record in the result set.

Because **DECLARE** is a declaration, not an executed statement, it does not set or kill the SQLCODE variable.

Cursor Names

Cursor names are case-sensitive.

A cursor name must be unique within the routine and the corresponding class. A cursor name may be of any length, but must be unique within the first 29 characters. Cursor names are case-sensitive. If a specified cursor has already been declared, no compilation error is issued; SQL execution uses the most recently declared instance of that cursor.

Cursor names are not namespace-specific. You can **DECLARE** a cursor in one namespace, and **OPEN**, **FETCH**, or **CLOSE** this cursor when in another namespace. Embedded SQL is complied when the **OPEN** command is executed. SQL tables and local variables are namespace-specific, so the **OPEN** operation must be invoked in the same namespace (or be able to access tables in the namespace) where the table(s) specified in the query are located.

The first character of a cursor name must be a letter. The second and subsequent characters of a cursor name must be either a letter or a number. Unlike SQL identifiers, punctuation characters are not permitted in cursor names.

You can use a delimiter characters (double quotes) to specify an SQL reserved word as a cursor name. A delimited cursor name is *not* an SQL delimited identifier; delimited cursor names are still case-sensitive and cannot contain punctuation characters. In most cases, an SQL reserved word should not be used as a cursor name.

Updating through a Cursor

You can perform record updates and deletes through a declared cursor using an **UPDATE** or **DELETE** statement with the WHERE CURRENT OF clause. In InterSystems SQL a cursor can always be used for **UPDATE** or **DELETE** operations if you have the appropriate privileges on the affected tables and columns; refer to the GRANT statement for assigning object privileges.

A **DECLARE** statement can specify a FOR UPDATE or FOR READ ONLY keyword clause following the query. These clauses are optional and perform no operation. They are provided as a way to document in the code that the process issuing the query has or does not have the needed update and delete object privileges.

Arguments

cursor-name

The name of the cursor, which must begin with a letter and contain only letters and numbers. (Cursor names do not follow SQL identifier conventions). Cursor names are case-sensitive. They are subject to additional naming restrictions, as described below.

query

A standard SELECT statement that defines the result set of the cursor. This **SELECT** can include the %NOFPLAN keyword to specify that InterSystems IRIS should ignore the frozen plan (if any) for this query. This **SELECT** can include an ORDER BY clause, with or without a TOP clause. This **SELECT** can specify a table-valued function in the **FROM** clause.

Examples

The following Embedded SQL example uses **DECLARE** to define a cursor for a query that specifies two output host variables. The cursor is then opened, fetched repeatedly, and closed:

ObjectScript

```
SET name="John Doe",state="##"
&sql(DECLARE EmpCursor CURSOR FOR
     SELECT Name, Home_State
     INTO :name,:state FROM Sample.Person
     WHERE Home_State %STARTSWITH 'A'
     FOR READ ONLY)
  WRITE !,"BEFORE: Name=",name," State=",state
&sql(OPEN EmpCursor)
    IF SQLCODE<0 {WRITE "SQL Open Cursor Error:",SQLCODE," ",%msg  QUIT}
    NEW %ROWCOUNT,%ROWID
FOR { &sql(FETCH EmpCursor)
    QUIT:SQLCODE
    WRITE !,"DURING: Name=",name," State=",state }
WRITE !,"FETCH status SQLCODE=",SQLCODE
WRITE !,"Number of rows fetched=",%ROWCOUNT
&sql(CLOSE EmpCursor)
    IF SQLCODE<0 {WRITE "SQL Close Cursor Error:",SQLCODE," ",%msg  QUIT}
WRITE !,"AFTER: Name=",name," State=",state
```

The following Embedded SQL example uses **DECLARE** to define a cursor for a query that specifies both output host variables in the INTO clause and input host variables in the WHERE clause. The cursor is then opened, fetched repeatedly, and closed:

ObjectScript

```
NEW SQLCODE,%ROWCOUNT,%ROWID
SET EmpZipLow="10000"
SET EmpZipHigh="19999"
 &sql(DECLARE EmpCursor CURSOR FOR
   SELECT Name,Home_Zip
   INTO :name,:zip
   FROM Sample.Employee WHERE Home_Zip BETWEEN :EmpZipLow AND :EmpZipHigh)
&sql(OPEN EmpCursor)
   IF SQLCODE<0 {WRITE "SQL Open Cursor Error:",SQLCODE," ",%msg  QUIT}
 FOR { &sql(FETCH EmpCursor)
     QUIT:SQLCODE
     WRITE !,name," ",zip }
 &sql(CLOSE EmpCursor)
   IF SQLCODE<0 {WRITE "SQL Close Cursor Error:",SQLCODE," ",%msg  QUIT}
```

The following Embedded SQL example uses a table-valued function as the **FROM** clause of the *query*:

ObjectScript

```
SET $NAMESPACE="Samples"
&sql(DECLARE EmpCursor CURSOR FOR
     SELECT Name INTO :name FROM Sample.SP_Sample_By_Name('A')
     FOR READ ONLY)
&sql(OPEN EmpCursor)
 IF SQLCODE<0 {WRITE "SQL Open Cursor Error:",SQLCODE," ",%msg  QUIT}
NEW %ROWCOUNT,%ROWID
FOR { &sql(FETCH EmpCursor)
     QUIT:SQLCODE
     WRITE "Name=",name,! }
WRITE !,"FETCH status SQLCODE=",SQLCODE
WRITE !,"Number of rows fetched=",%ROWCOUNT
&sql(CLOSE EmpCursor)
 IF SQLCODE<0 {WRITE "SQL Close Cursor Error:",SQLCODE," ",%msg  QUIT}
```

See Also

- CLOSE command

- FETCH command

- OPEN command

- WHERE CURRENT OF clause

- SQL Cursors

DELETE (SQL)

Removes rows from a table.

Synopsis

```
DELETE [%keyword] [FROM] table-ref [[AS] t-alias]
    [FROM [optimize-option] select-table [[AS] t-alias]
    {,select-table2 [[AS] t-alias]} ]
    [WHERE condition-expression]

DELETE [%keyword] [FROM] table-ref [[AS] t-alias]
    [WHERE CURRENT OF cursor]
```

Arguments

Argument	Description
%keyword	*Optional* — One or more of the following keyword options, separated by spaces: %NOCHECK, %NOFPLAN, %NOINDEX, %NOJOURN, %NOLOCK, %NOTRIGGER, %PROFILE, %PROFILE_ALL.
FROM *table-ref*	The table from which you are deleting rows. This is *not* a FROM clause; it is a FROM keyword followed by a single table reference. (The FROM keyword is optional; the *table-ref* is mandatory.)
	A table name (or view name) can be qualified (schema.table), or unqualified (table). An unqualified name is matched to its schema using either a schema search path (if provided) or the default schema name.
	Rather than a table reference, you can specify a view through which table rows can be deleted, or specify a subquery enclosed in parentheses. Unlike the **SELECT** statement FROM clause, you cannot specify *optimize-option* keywords here. You cannot specify a table-valued function or JOIN syntax in this argument.
FROM clause	*Optional* — A FROM clause, specified *after* the *table-ref*. This FROM can be used to specify a *select-table* table or tables used to select which rows are to be deleted.
	Multiple tables can be specified as a comma-separated list or associated with ANSI join keywords. Any combination of tables or views can be specified. If you specify a comma between two *select-table*s here, InterSystems IRIS performs a CROSS JOIN on the tables and retrieves data from the results table of the JOIN operation. If you specify ANSI join keywords between two *select-table*s here, InterSystems IRIS performs the specified join operation. For further details, see JOIN.
	You can optionally specify one or more *optimize-option* keywords to optimize queryexecution. The available options are: %ALLINDEX, %FIRSTTABLE *tablename*, %FULL, %INORDER, %IGNOREINDICES, %NOFLATTEN, %NOMERGE, %NOSVSO, %NOTOPOPT, %NOUNIONOROPT, %PARALLEL, and %STARTTABLE. See FROM clause for more details.

Argument	Description
AS *t-alias*	*Optional* — An alias for a table or view name. An alias must be a valid identifier. The AS keyword is optional.
WHERE *condition-expression*	*Optional* — Specifies one or more boolean predicates used to limit which rows are to be deleted. You can specify a **WHERE** clause or a **WHERE CURRENT OF** clause, but not both. If a **WHERE** clause (or a **WHERE CURRENT OF** clause) is not supplied, **DELETE** removes all the rows from the table. For further details, see WHERE.
WHERE CURRENT OF *cursor*	*Optional: Embedded SQL only* — Specifies that the **DELETE** operation deletes the record at the current position of cursor. You can specify a **WHERE CURRENT OF** clause or a **WHERE** clause, but not both. If a **WHERE CURRENT OF** clause (or a **WHERE** clause) is not supplied, **DELETE** removes all the rows from the table. For further details, see WHERE CURRENT OF.

Description

The **DELETE** command removes rows from a table that meet the specified conditions. You can delete rows from a table directly, delete through a view, or delete rows selected using a subquery. Deleting through a view is subject to requirements and restrictions, as described in CREATE VIEW.

The **DELETE** operation sets the %ROWCOUNT local variable to the number of deleted rows, and the %ROWID local variable to the RowID value of the last row deleted. If no rows are deleted, %ROWCOUNT=0 and %ROWID is undefined or remains set to its previous value.

You must specify a *table-ref*; the FROM keyword before the *table-ref* is optional. To delete *all* rows from a table, you can simply specify:

SQL

```
DELETE FROM tablename
```

or

SQL

```
DELETE tablename
```

This deletes all row data from the table, but does not reset the RowID, IDENTITY, stream field OID values, and SERIAL (%Library.Counter) field counters. The TRUNCATE TABLE command both deletes all row data from a table and resets these counters. By default, `DELETE FROM tablename` pulls delete triggers; you can speciy `DELETE %NOTRIGGER FROM tablename` to not pull delete triggers. **TRUNCATE TABLE** does not pull delete triggers.

More commonly, a **DELETE** specifies the deletion of a specific row (or rows) based on a *condition-expression*. By default, a **DELETE** operation goes through all of the rows of a table and deletes all rows that satisfy the *condition-expression*. If no rows satisfy the *condition-expression*, **DELETE** completes successfully and sets SQLCODE=100 (No more data).

You can specify a **WHERE** clause or a **WHERE CURRENT OF** clause (but not both). If the **WHERE CURRENT OF** clause is used, the **DELETE** operation deletes the record at the current position of the cursor. For an example of **DELETE** using **WHERE CURRENT OF**, see Embedded SQL and Dynamic SQL Examples below. For details on positioned operations, see WHERE CURRENT OF.

By default, **DELETE** is an all-or-nothing event: either all specified rows are deleted completely, or no deletion is performed. InterSystems IRIS sets the status variable SQLCODE, indicating the success or failure of the **DELETE**.

To delete a row from a table:

- The table must exist in the current (or specified) namespace. If the specified table cannot be located, InterSystems IRIS issues an SQLCODE -30 error.

- The user must have DELETE privilege on the specified table. If the user is the Owner (creator) of the table, the user is automatically granted DELETE privilege for that table. Otherwise, the user must be granted DELETE privilege for the table. Failing to do so results in an SQLCODE –99 error with the %msg User 'name' is not privileged for the operation. You can determine if the current user has DELETE privilege by invoking the %CHECKPRIV command. You can use the GRANT command to assign DELETE privilege to a specified table. For further details, refer to Privileges.

- The table cannot be locked IN EXCLUSIVE MODE by another process. Attempting to delete a row from a locked table results in an SQLCODE -110 error, with a %msg such as the following: Unable to acquire lock for DELETE of table 'Sample.Person' on row with RowID = '10'. Note that an SQLCODE -110 error occurs only when the **DELETE** statement locates the first record to be deleted, then cannot lock it within the timeout period.

- If the **DELETE** command's WHERE clause specifies a non-existent field, an SQLCODE -29 is issued. To list all of the field names defined for a specified table, refer to Column Names and Numbers. If the field exists but none of the field values fulfill the **DELETE** command's WHERE clause, no rows are affected and SQLCODE 100 (end of data) is issued.

- The table cannot be defined as READONLY. Attempting to compile a **DELETE** that references a read-only table results in an SQLCODE -115 error. Note that this error is now issued at compile time, rather than only occurring at execution time. See the description of READONLY objects in Other Options for Persistent Classes.

- If deleting through a view, the view cannot be defined as WITH READ ONLY. Attempting to do so results in an SQLCODE -35 error. If the view is based on a sharded table, you cannot **DELETE** through a view defined WITH CHECK OPTION. Attempting to do so results in an SQLCODE -35 with the %msg INSERT/UPDATE/DELETE not allowed for view (sample.myview) based on sharded table with check option conditions. See the CREATE VIEW command for further details. Similarly, if you are attempting to delete through a subquery, the subquery must be updateable; for example, the following subquery results in an SQLCODE -35 error: DELETE FROM (SELECT COUNT(*) FROM Sample.Person) AS x.

- The row to delete must exist. Usually, attempting to delete a nonexistent row results in an SQLCODE 100 (No more data) because the specified row could not be located. However, in rare cases, **DELETE** with %NOLOCK locates a row to be deleted, but then the row is immediately deleted by another process; this situation results in an SQLCODE -106 error. The %msg for this error lists the table name and the RowID.

- All of the rows specified for deletion must be available for deletion. By default, if one or more rows cannot be deleted the **DELETE** operation fails and no rows are deleted. If a row to be deleted has been locked by another concurrent process, **DELETE** issues an SQLCODE -110 error. If deleting one of the specified rows would violate foreign key referential integrity (and %NOCHECK is not specified), the **DELETE** issues an SQLCODE -124 error. This default behavior is modifiable, as described below.

- Certain %SYS namespace system–supplied facilities are protected against deletion. For example, DELETE FROM Security.Users cannot be used to delete _SYSTEM, _PUBLIC or UnknownUser. Attempting to do so results in an SQLCODE -134 error.

FROM Syntax

A **DELETE** command can contain two FROM keywords that specify tables. These two uses of FROM are fundamentally different:

- FROM before *table-ref* specifies the table (or view) from which rows are to be deleted. It is a FROM keyword, not a FROM clause. Only one table may be specified. No join syntax or *optimize-option* keywords may be specified. The FROM keyword itself is optional; the *table-ref* is required.

- FROM after *table-ref* is an optional FROM clause that can be used to determine which rows should be deleted. It may specify one or more than one tables. It supports all of the FROM clause syntax available to a SELECT statement, including join syntax and *optimize-option* keywords. This FROM clause is commonly (but not always) used with a WHERE clause.

Thus any of the following are valid syntactical forms:

```
DELETE FROM table WHERE ...   DELETE table WHERE ...   DELETE
FROM table FROM table2 WHERE ...   DELETE table FROM table2 WHERE ...
```

This syntax supports complex selection criteria in a manner compatible with Transact-SQL.

The following example shows how the two FROM keywords might be used. It deletes those records from the Employees table where the same EmpId is also found in the Retirees table:

SQL

```
DELETE FROM Employees AS Emp
      FROM Retirees AS Rt
      WHERE Emp.EmpId = Rt.EmpId
```

If the two FROM keywords make reference to the same table, these references may either be to the same table, or to a join of two instances of the table. This depends on how table aliases are used:

- If neither table reference has an alias, both reference the same table:

    ```
    DELETE FROM table1 FROM table1,table2   /* join of 2 tables */
    ```

- If both table references have the same alias, both reference the same table:

    ```
    DELETE FROM table1 AS x FROM table1 AS x,table2   /* join of 2 tables */
    ```

- If both table references have aliases, and the aliases are different, InterSystems IRIS performs a join of two instances of the table:

    ```
    DELETE FROM table1 AS x FROM table1 AS y,table2   /* join of 3 tables */
    ```

- If the first table reference has an alias, and the second does not, InterSystems IRIS performs a join of two instances of the table:

    ```
    DELETE FROM table1 AS x FROM table1,table2   /* join of 3 tables */
    ```

- If the first table reference does not have an alias, and the second has a single reference to the table with an alias, both reference the same table, and this table has the specified alias:

    ```
    DELETE FROM table1 FROM table1 AS x,table2   /* join of 2 tables */
    ```

- If the first table reference does not have an alias, and the second has more than one reference to the table, InterSystems IRIS considers each aliased instance a separate table and performs a join on these tables:

    ```
    DELETE FROM table1 FROM table1,table1 AS x,table2   /* join of 3 tables */
    DELETE FROM table1 FROM table1 AS x,table1 AS y,table2   /* join of 4 tables */
    ```

%Keyword Options

Specifying *%keyword* argument(s) restricts processing as follows:

- %NOCHECK — suppress referential integrity checking for foreign keys that reference the rows being deleted. The user must have the corresponding %NOCHECK administrative privilege for the current namespace to apply this

restriction. Failing to do so results in an SQLCODE –99 error with the %msg `User 'name' does not have %NOCHECK privileges.`

- %NOFPLAN — the frozen plan (if any) is ignored for this operation; the operation generates a new query plan. The frozen plan is retained, but not used. For further details, refer to Frozen Plans.

- %NOINDEX — suppresses deleting index entries in all indexes for the rows being deleted. This should be used with extreme caution, because it leaves orphaned values in the table indexes. The user must have the corresponding %NOINDEX administrative privilege for the current namespace to apply this restriction. Failing to do so results in an SQLCODE –99 error with the %msg `User 'name' does not have %NOINDEX privileges.`

- %NOJOURN — suppress journaling and disable transactions for the duration of the delete operation. None of the changes made in any of the rows are journaled, including any triggers pulled. If you perform a **ROLLBACK** after a statement with %NOJOURN, the changes made by the statement will not be rolled back. The user must have the corresponding %NOJOURN administrative privilege for the current namespace to apply this restriction. Failing to do so results in an SQLCODE –99 error with the %msg `User 'name' does not have %NOJOURN privileges.`

- %NOLOCK — suppress row locking of the row being deleted. This should only be used when a single user/process is updating the database. The user must have the corresponding %NOLOCK administrative privilege for the current namespace to apply this restriction. Failing to do so results in an SQLCODE –99 error with the %msg `User 'name' does not have %NOLOCK privileges.`

- %NOTRIGGER — suppress the pulling of base table triggers that are otherwise pulled during **DELETE** processing. The user must have the corresponding %NOTRIGGER administrative privilege for the current namespace to apply this restriction. Failing to do so results in an SQLCODE –99 error with the %msg `User 'name' does not have %NOTRIGGER privileges.`

- %PROFILE or %PROFILE_ALL — if one of these keyword directives is specified, SQLStats collecting code is generated. This is the same code that would be generated with PTools turned ON. The difference is that SQLStats collecting code is only generated for this specific statement. All other SQL statements within the routine/class being compiled will generate code as if PTools is turned OFF. This enables the user to profile/inspect specific problem SQL statements within an application without collecting irrelevant statistics for SQL statements that are not being investigated. For further details, refer to SQL Runtime Statistics.

  %PROFILE collects SQLStats for the main query module. %PROFILE_ALL collects SQLStats for the main query module and all of its subquery modules.

You can specify multiple *%keyword* arguments in any order. Multiple arguments are separated by spaces.

If you specify a *%keyword* argument when deleting a parent record, the same *%keyword* argument will be applied when deleting the corresponding child records.

Referential Integrity

If you do not specify %NOCHECK, InterSystems IRIS uses the system-wide configuration setting to determine whether to perform foreign key referential integrity checking; the default is to perform foreign key referential integrity checking. You can set this default system-wide, as described in Foreign Key Referential Integrity Checking. To determine the current system-wide setting, call **$SYSTEM.SQL.CurrentSettings()**.

During a **DELETE** operation, for every foreign key reference a shared lock is acquired on the corresponding row in the referenced table. This row is locked until the end of the transaction. This ensures that the referenced row is not changed before a potential rollback of the **DELETE**.

If a series of foreign key references are defined as CASCADE, a **DELETE** operation could potentially result in a circular reference. InterSystems IRIS prevents **DELETE** with CASCADE referential action from performing a circular reference loop recursion. InterSystems IRIS ends the cascade sequence when it returns to the original table.

If a **DELETE** operation with %NOLOCK is performed on a foreign key field defined with CASCADE, SET NULL, or SET DEFAULT, the corresponding referential action changing the foreign key table is also performed with %NOLOCK.

Atomicity

By default, **DELETE**, **UPDATE**, **INSERT**, and **TRUNCATE TABLE** are atomic operations. A **DELETE** either completes successfully or the whole operation is rolled back. If any of the specified rows cannot be deleted, none of the specified rows are deleted and the database reverts to its state before issuing the **DELETE**.

You can modify this default for the current process within SQL by invoking SET TRANSACTION %COMMITMODE. You can modify this default for the current process in ObjectScript by invoking the **SetOption()** method, as follows SET status=$SYSTEM.SQL.Util.SetOption("AutoCommit",intval,.oldval). The following *intval* integer options are available:

- 1 or IMPLICIT (autocommit on) — The default behavior, as described above. Each **DELETE** constitutes a separate transaction.

- 2 or EXPLICIT (autocommit off) — If no transaction is in progress, a **DELETE** automatically initiates a transaction, but you must explicitly **COMMIT** or **ROLLBACK** to end the transaction. In EXPLICIT mode the number of database operations per transaction is user-defined.

- 0 or NONE (no auto transaction) — No transaction is initiated when you invoke **DELETE**. A failed **DELETE** operation can leave the database in an inconsistent state, with some of the specified rows deleted and some not deleted. To provide transaction support in this mode you must use **START TRANSACTION** to initiate the transaction and **COMMIT** or **ROLLBACK** to end the transaction.

  A sharded table is always in no auto transaction mode, which means all inserts, updates, and deletes to sharded tables are performed outside the scope of a transaction.

You can determine the atomicity setting for the current process using the **GetOption("AutoCommit")** method, as shown in the following ObjectScript example:

ObjectScript

```
SET stat=$SYSTEM.SQL.Util.SetOption("AutoCommit",$RANDOM(3),.oldval)
IF stat'=1 {WRITE "SetOption failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET x=$SYSTEM.SQL.Util.GetOption("AutoCommit")
  IF x=1 {
    WRITE "Default atomicity behavior",!
    WRITE "automatic commit or rollback" }
  ELSEIF x=0 {
    WRITE "No transaction initiated, no atomicity:",!
    WRITE "failed DELETE can leave database inconsistent",!
    WRITE "rollback is not supported" }
  ELSE { WRITE "Explicit commit or rollback required" }
```

Transaction Locking

If you do not specify %NOLOCK, the system automatically performs standard record locking on **INSERT**, **UPDATE**, and **DELETE** operations. Each affected record (row) is locked for the duration of the current transaction.

The default lock threshold is 1000 locks per table, which means if you delete more than 1000 records from a table during a transaction, the lock threshold is reached and InterSystems IRIS automatically escalates the locking level from record locks to a table lock. This permits large-scale deletes during a transaction without overflowing the lock table.

InterSystems IRIS applies one of the two following lock escalation strategies:

- "E"-type lock escalation: InterSystems IRIS uses this type of lock escalation if the following are true: (1) the class uses %Storage.Persistent (you can determine this from the Catalog Details in the Management Portal SQL schema display). (2) the class either does not specify an IDKey index, or specifies a single-property IDKey index. "E"-type lock escalation is described in the LOCK command.

- Traditional SQL lock escalation: The most likely reason why a class would not use "E"-type lock escalation is the presence of a multi-property IDKey index. In this case, each %Save increments the lock counter. This means if you do 1001 saves of a single object within a transaction, InterSystems IRIS will attempt to escalate the lock.

For both lock escalation strategies, you can determine the current system-wide lock threshold value using the **$SYSTEM.SQL.Util.GetOption("LockThreshold")** method. The default is 1000. This system-wide lock threshold value is configurable:

- Using the **$SYSTEM.SQL.Util.SetOption("LockThreshold")** method.

- Using the Management Portal: select **System Administration**, **Configuration**, **SQL and Object Settings**, **SQL**. View and edit the current setting of **Lock escalation threshold**. The default is 1000 locks. If you change this setting, any new process started after changing it will have the new setting.

You must have USE permission on the %Admin Manage Resource to change the lock threshold. InterSystems IRIS immediately applies any change made to the lock threshold value to all current processes.

One potential consequence of automatic lock escalation is a deadlock situation that might occur when an attempt to escalate to a table lock conflicts with another process holding a record lock in that table. There are several possible strategies to avoid this: (1) increase the lock escalation threshold so that lock escalation is unlikely to occur within a transaction. (2) substantially lower the lock escalation threshold so that lock escalation occurs almost immediately, thus decreasing the opportunity for other processes to lock a record in the same table. (3) apply a table lock for the duration of the transaction and do not perform record locks. This can be done at the start of the transaction by specifying LOCK TABLE, then UNLOCK TABLE (without the IMMEDIATE keyword, so that the table lock persists until the end of the transaction), then perform deletes with the %NOLOCK option.

Automatic lock escalation is intended to prevent overflow of the lock table. However, if you perform such a large number of deletes that a <LOCKTABLEFULL> error occurs, **DELETE** issues an SQLCODE -110 error.

For further details on transaction locking refer to Transaction Processing.

Examples

The following examples both delete all rows from the TempEmployees table. Note that the FROM keyword is optional:

SQL

```
DELETE FROM TempEmployees
```

SQL

```
DELETE TempEmployees
```

The following example deletes employee number 234 from the Employees table:

SQL

```
DELETE
    FROM Employees
    WHERE EmpId = 234
```

The following example deletes all rows from the ActiveEmployees table in which the CurStatus column is set to "Retired":

SQL

```
DELETE FROM ActiveEmployees
    WHERE CurStatus = 'Retired'
```

The following example deletes rows using a subquery:

SQL

```
DELETE FROM (SELECT Name,Age FROM Sample.Person WHERE Age > 65)
```

Table Deletion Example

The following example demonstrates the task of deleting rows from a newly-created table and then subsequetly deleting the table itself.

The first command in this example creates a table named SQLUser.WordPairs with three columns.

SQL

```
CREATE TABLE SQLUser.WordPairs (
    Lang        CHAR(2) NOT NULL,
    Firstword   CHAR(30),
    Lastword    CHAR(30))
```

The next few commands insert six records into the table.

SQL

```
INSERT INTO WordPairs (Lang,Firstword,Lastword) VALUES
    ('En','hello','goodbye')
INSERT INTO WordPairs (Lang,Firstword,Lastword) VALUES
    ('Fr','bonjour','au revoir')
INSERT INTO WordPairs (Lang,Firstword,Lastword) VALUES
    ('It','pronto','ciao')
INSERT INTO WordPairs (Lang,Firstword,Lastword) VALUES
    ('Fr','oui','non')
INSERT INTO WordPairs (Lang,Firstword,Lastword) VALUES
    ('En','howdy','see ya')
INSERT INTO WordPairs (Lang,Firstword,Lastword) VALUES
    ('Es','hola','adios')
```

The following commands delete all English records using cursor-based Embedded SQL.

ObjectScript

```
#sqlcompile path=Sample
  NEW %ROWCOUNT,%ROWID
  &sql(DECLARE WPCursor CURSOR FOR
       SELECT Lang FROM WordPairs
       WHERE Lang='En')
  &sql(OPEN WPCursor)
       QUIT:(SQLCODE'=0)
  FOR { &sql(FETCH WPCursor)
       QUIT:SQLCODE
       &sql(DELETE FROM WordPairs
       WHERE CURRENT OF WPCursor)
   IF SQLCODE=0 {
   WRITE !,"Delete succeeded"
   WRITE !,"Row count=",%ROWCOUNT," RowID=",%ROWID }
   ELSE {
   WRITE !,"Delete failed, SQLCODE=",SQLCODE }
   }
       &sql(CLOSE WPCursor)
```

This command then deletes all French records.

SQL

```
DELETE FROM WordPairs WHERE Lang='Fr'
```

The final two commands display the remaining records in the table and delete the table.

SQL

```
SELECT %ID,* FROM SQLUser.WordPairs
DROP TABLE SQLUser.WordPairs
```

See Also

- FROM
- TRUNCATE TABLE
- INSERT UPDATE
- CREATE VIEW
- WHERE
- WHERE CURRENT OF
- Modifying the Database
- Defining Tables
- Defining Views
- Transaction Processing
- SQL and Object Settings Pages
- SQLCODE error messages

DROP AGGREGATE (SQL)

Deletes a user-defined aggregate function.

Synopsis

```
DROP AGGREGATE [IF EXISTS] name
```

Description

The **DROP AGGREGATE** command deletes a user-defined aggregate function (UDAF). A user-defined aggregate function is created using the CREATE AGGREGATE command.

If you attempt to drop a UDAF that does not exist, SQL issues an SQLCODE -428 error, with a message such as: `User Defined Aggregate Function Sample.SecondHighest does not exist.`

Dropping a UDAF automatically purges any cached queries that reference that UDAF.

Arguments

name

The name of the user-defined aggregate function to be deleted. The *name* can be qualified (schema.aggname), or unqualified (aggname). An unqualified *name* takes the default schema name.

See Also

- CREATE AGGREGATE command

- Overview of Aggregate Functions

- SQLCODE error messages

DROP DATABASE (SQL)

Deletes a database (namespace).

Synopsis

```
DROP DATABASE [IF EXISTS] dbname [RETAIN_FILES]
```

Description

The **DROP DATABASE** command deletes a namespace and its associated database.

The specified *dbname* is the name of the namespace and the directory that contains the corresponding database files. Specify *dbname* as an identifier. Namespace names are not case-sensitive. If the specified *dbname* namespace does not exist, InterSystems IRIS issues an SQLCODE -340 error.

The **DROP DATABASE** command is a privileged operation. Prior to using **DROP DATABASE**, it is necessary to be logged in as a user with the %Admin_Manage resource. The user must also have READ permission on the resource for the routines and global's database definitions. Failing to do so results in an SQLCODE -99 error (Privilege Violation).

Use the **$SYSTEM.Security.Login()** method to assign a user with appropriate privileges:

ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql(        )
```

You must have the **`%Service_Login:Use`** privilege to invoke the **$SYSTEM.Security.Login** method. For further information, see %SYSTEM.Security.

DROP DATABASE cannot be used to drop a system namespace, regardless of privileges. Attempting to do so results in an SQLCODE -342 error.

DROP DATABASE cannot be used to drop the namespace that you are currently using or connected to. Attempting to do so results in an SQLCODE -344 error.

You can also delete a namespace using the Management Portal. Select **System Administration**, **Configuration**, **System Configuration**, **Namespaces** to list the existing namespaces. Click the **Delete** button for the namespace you wish to delete.

RETAIN_FILES

If you specify this option, the physical file structure is retained; the database and its associated namespace is removed. After performing this operation, a subsequent attempt to use *dbname* results in the following:

- **DROP DATABASE** without RETAIN_FILES *cannot* remove this physical file structure. Instead, it results in an SQLCODE -340 error (Database not found).

- **DROP DATABASE** with RETAIN_FILES also results in an SQLCODE -340 error (Database not found).

- **CREATE DATABASE** *cannot* create a new database with the same name. Instead, it results in an SQLCODE -341 error (Cannot create database file for database).

- Attempting to use this namespace results in a <NAMESPACE> error.

Server Init and Disconnect Codes

The Server Init Code and Server Disconnect Code can be assigned to a namespace using the **$SYSTEM.SQL.Util.SetOption("ServerInitCode",value)** and **$SYSTEM.SQL.Util.SetOption("ServerDisconnectCode",value)** methods. The current values can be determined using the corresponding **$SYSTEM.SQL.Util.GetOption()** method options.

Deleting a namespace, using **DROP DATABASE** or other interfaces, deletes these Server Init Code and Server Disconnect Code values. Therefore, deleting and then re-creating a namespace will require you to re-specify these values.

Arguments

IF EXISTS

An optional argument that, if specified, suppresses the error if the command is executed on a nonexistent database.

dbname

The name of the database (namespace) to be deleted.

RETAIN_FILES

An optional argument that, if specified, the physical database files (IRIS.DAT files) will not be deleted. The default is to delete the .DAT files along with the namespace and the other database entities.

Example

The following example deletes a namespace and its associated database (in this case 'c:\InterSystems\IRIS\mgr\DocTestDB'). It retains the physical database files:

SQL

```
CREATE DATABASE DocTestDB ON DIRECTORY 'c:\InterSystems\IRIS142\mgr\DocTestDB'
```

SQL

```
DROP DATABASE DocTestDB RETAIN_FILES
```

See Also

- CREATE DATABASE command
- USE DATABASE command

DROP FOREIGN SERVER (SQL)

Drops a foreign server.

Synopsis

```
DROP [ FOREIGN ] SERVER server-name [ RESTRICT | CASCADE ]
```

Arguments

Arguments	Description
server-name	The name of the foreign server to be dropped. This name must be a valid identifier. A foreign server by this name must exist for the command to execute successfully.
RESTRICT	*Optional* — Specifies that the foreign server should be dropped if nothing is defined on it. This option supplies the default behavior.
CASCADE	*Optional* — Specifies that all objects defined within the foreign server, including tables, are dropped with the foreign server.

Description

The DROP FOREIGN SERVER command deletes a foreign server that was configured to host foreign tables.

By default, this command will only drop a foreign server that has no foreign tables defined on it; you may explicitly apply this behavior by specifying the RESTRICT keyword. When the RESTRICT keyword has been implicitly or explicitly specified, attempting to delete a foreign server with at least one table defined on it generates an SQLCODE -321 error.

When the CASCADE is specified, DROP FOREIGN SERVER will successfully delete the foreign server and all of the tables defined on it.

In order to delete a foreign table, the following conditions must be met:

- The foreign server must exist in the current namespace. Attempting to delete a non-existent foreign server generates an SQLCODE -30 error.

- You must have the necessary privileges to delete the foreign server. Attempting to delete a foreign server without the necessary privileges generates an SQLCODE -99 error.

Examples

The following example drops a foreign server that does not have any tables defined on it.

```
DROP FOREIGN SERVER EmptyServer RESTRICT
```

The following example drops a foreign server that has tables defined on it. In the process of dropping the foreign server, the tables associated with it are also dropped.

```
DROP FOREIGN SERVER FullServer CASCADE
```

See Also

- CREATE FOREIGN SERVER

- ALTER FOREIGN SERVER

- DROP FOREIGN TABLE

DROP FOREIGN TABLE (SQL)

Drops a foreign table.

Synopsis

```
DROP FOREIGN TABLE [ IF EXISTS ] table-name [ RESTRICT | CASCADE ]
```

Arguments

Arguments	Description
IF EXISTS	*Optional* — Suppresses the error that arises if a foreign table with *table-name* does not exist.
table-name	The name of the foreign table to be dropped. This name must be a valid identifier. A foreign table by this name must exist for the command to execute successfully.
RESTRICT	*Optional* — Specifies that the foreign table should not be dropped if any SQL objects, such as views, are defined on the foreign table. This option supplies the default behavior.
CASCADE	*Optional* — Specifies that all objects defined on the foreign table, such as views, are dropped with the foreign table.

Description

The DROP FOREIGN TABLE command deletes a foreign table from a foreign server.

By default, this command will only delete a foreign table if no views are associated with it; you may explicitly apply this behavior by specifying the RESTRICT keyword. When the RESTRICT keyword has been implicitly or explicitly specified, attempting to delete a foreign table with associated views generates an SQLCODE -321 error.

When the CASCADE keyword is specified, DROP FOREIGN TABLE will successfully delete the table and any views associated with it.

In order to delete a foreign table, the following conditions must be met:

- The foreign table must exist on a foreign server in the current namespace. Attempting to delete a non-existent foreign table generates an SQLCODE -30 error. This error is suppressed by epcifying the IF EXISTS keywords.

- You must have the necessary privileges to delete the foreign table. Attempting to delete a table without the necessary privileges generates an SQLCODE -99 error.

Examples

The following example deletes a foreign table that does not have any objects defined on it.

```
DROP FOREIGN TABLE Example.MyTable RESTRICT
```

The following example deletes a foreign table and any views associated with it.

```
DROP FOREIGN TABLE Example.MyTable CASCADE
```

See Also

- CREATE FOREIGN TABLE

- [ALTER FOREIGN SERVER](#)
- [DROP FOREIGN SERVER](#)

DROP FUNCTION (SQL)

Deletes a function.

Synopsis

```
DROP FUNCTION [IF EXISTS] name [ FROM className ]
```

Description

The **DROP FUNCTION** command deletes a function. When you drop a function, InterSystems IRIS revokes it from all users and roles to whom it has been granted and removes it from the database.

In order to drop a function, you must have %DROP_FUNCTION administrative privilege, as specified by the GRANT command. Otherwise, the system generates an SQLCODE -99 error (Privilege Violation).

You cannot drop a function if the class definition that contains that function definition is a deployed class. This operation fails with an SQLCODE -400 error with the %msg `Unable to execute DDL that modifies a deployed class: 'classname'`.

The following combinations of *name* and FROM *className* are supported. Note that the FROM clause specifies the class package name and function name, not the SQL names. In these examples, the system-wide default schema name is SQLUser, which corresponds to the User class package:

- `DROP FUNCTION BonusCalc FROM funcBonusCalc`: drops the function SQLUser.BonusCalc().

- `DROP FUNCTION BonusCalc FROM User.funcBonusCalc`: drops the function SQLUser.BonusCalc().

- `DROP FUNCTION Test.BonusCalc FROM funcBonusCalc`: drops the function SQLUser.BonusCalc().

- `DROP FUNCTION BonusCalc FROM Employees.funcBonusCalc`: drops the function Employees.BonusCalc().

- `DROP FUNCTION Test.BonusCalc FROM Employees.funcBonusCalc`: drops the function Employees.Bonus-Calc().

If the specified function does not exist, **DROP FUNCTION** generates an SQLCODE -362 error. If the specified class does not exist, **DROP FUNCTION** generates an SQLCODE -360 error. If the specified function could refer to two or more functions, **DROP FUNCTION** generates an SQLCODE -361 error; you must specify a *className* to resolve this ambiguity.

Arguments

IF EXISTS

An optional argument that suppresses the error if the command is executed on a nonexistent function.

name

The name of the function to be deleted. The name is an identifier. Do not specify the function's parameter parentheses. A *name* can be qualified (schema.name), or unqualified (name). An unqualified function name takes the system-wide default schema name, unless the FROM *className* clause is specified.

FROM className

If specified, the **FROM** *className* clause deletes the function from the given class. Note that you must specify the className of a function (funcBonusCalc), not the SQL name (BonusCalc). If the **FROM** clause is not specified, InterSystems IRIS searches all classes of the schema for the function, and deletes it. However, if no function of this name is found, or more than one function of this name is found, an error code is returned. If the deletion of the function results in an empty class, **DROP FUNCTION** deletes the class as well

---

Examples

The following example attempts to delete myfunc from the class User.Employee. (Refer to **CREATE TABLE** for an example that creates class User.Employee.)

SQL

```
DROP FUNCTION myfunc FROM User.Employee
```

See Also

- CREATE FUNCTION
- SQLCODE error messages

DROP INDEX (SQL)

Removes an index.

Synopsis

```
DROP INDEX [IF EXISTS] [%NOJOURN] index-name [ON [TABLE] table-name]

DROP INDEX [IF EXISTS] table-name.index-name
```

Description

A **DROP INDEX** statement deletes an index from a table definition. You can use **DROP INDEX** to delete a standard index, bitmap index, or bitslice index. You can use **DROP INDEX** to delete a unique constraint or a primary key constraint by deleting the corresponding Unique index. You cannot use **DROP INDEX** to delete a Bitmap Extent index or a Master Map (Data/Master) IDKEY index.

You may wish to delete an index for any of the following reasons:

- You intend to perform large numbers of **INSERT**, **UPDATE**, or **DELETE** operations on a table. Rather than accepting the performance overhead of having each of these operations write to the index, you can use the %NOINDEX option for the operation. Or, in certain cases, it may be preferable to delete the index, perform the bulk changes to the database, and then recreate the index and populate it.

- An index exists for a field or combination of fields that are not used for query operations. In this case, the performance overhead of maintaining the index may not be worthwhile.

- An index exists for a field or combination of fields that now contain large amounts of duplicate data. In this case, the minimal gain to query performance may not be worthwhile.

You cannot drop an IDKEY index when there is data in the table. Attempting to do so generates an SQLCODE -325 error.

Privileges and Locking

The **DROP INDEX** command is a privileged operation. The user must have %ALTER_TABLE administrative privilege to execute **DROP INDEX**. Failing to do so results in an SQLCODE –99 error with the %msg User 'name' does not have %ALTER_TABLE privileges. You can use the GRANT command to assign %ALTER_TABLE privileges to a user or role, if you hold appropriate granting privileges. Administrative privileges are namespace-specific. For further details, refer to Privileges.

The user must have %ALTER privilege on the specified table. If the user is the Owner (creator) of the table, the user is automatically granted %ALTER privilege for that table. Otherwise, the user must be granted %ALTER privilege for the table. Failing to do so results in an SQLCODE –99 error with the %msg User 'name' does not have required %ALTER privilege needed to change the table definition for 'Schema.TableName'. You can determine if the current user has %ALTER privilege by invoking the %CHECKPRIV command. You can use the GRANT command to assign %ALTER privilege to a specified table. For further details, refer to Privileges.

- **DROP INDEX** cannot be used on a table projected from a persistent class, unless the table class definition includes [DdlAllowed]. Otherwise, the operation fails with an SQLCODE -300 error with the %msg DDL not enabled for class 'Schema.tablename'.

- **DROP INDEX** cannot be used on a table projected from a deployed persistent class. This operation fails with an SQLCODE -400 error with the %msg Unable to execute DDL that modifies a deployed class: 'classname'.

The **DROP INDEX** statement acquires a table-level lock on *table-name*. This prevents other processes from modifying the table's data. This lock is automatically released at the conclusion of the **DROP INDEX** operation.

Index Name

When specifying an *index-name* to create an index, the system generates a corresponding class index name by stripping out any punctuation characters; it retains the *index-name* you specified in the class as the SqlName value for the index (the SQL map name). When you specify an *index-name* to **DROP INDEX**, you specify the name including the punctuation, which is listed in the table's Management Portal SQL Catalog Details as the **SQL Map Name**. For example, you specify the generated **SQL Map Name** for a Unique constraint (MYTABLE_UNIQUE2), not the **Index Name** (MYTABLEUNIQUE2). This *index-name* is not case-sensitive.

Table Name

You can specify the table associated with the index using either **DROP INDEX** syntax form:

- `index-name ON TABLE` syntax: specifying the table name is optional. If omitted, InterSystems IRIS searches all of the classes in the namespace for the corresponding index.

- `table-name.index-name` syntax: specifying the table name is required.

In either syntax, the table name can be unqualified (table), or qualified (schema.table). If the schema name is omitted, the default schema name is used.

If **DROP INDEX** does not specify a table name, InterSystems IRIS searches through all indexes for an index SqlName matching *index-name*, or an index name matching *index-name* for indexes where an SqlName is not specified for the index. If InterSystems IRIS finds no matching indexes in any class, an SQLCODE -333 error is generated, indicating no such index exists. If InterSystems IRIS finds more than one matching index, **DROP INDEX** cannot determine which index to drop; it issues an SQLCODE -334 error: "Index name is ambiguous. Index found in multiple tables." Index names in InterSystems IRIS are not unique per namespace.

Nonexistent Index

By default, if you try to delete a nonexistent index, **DROP INDEX** issues an SQLCODE -333 error. To determine the current setting, call **$SYSTEM.SQL.CurrentSettings()**, which displays an `Allow DDL DROP of non-existent index` setting. The default is 0 ("No"). This is the recommended setting. If set to 1 ("Yes") **DROP INDEX** for a nonexistent index performs no operation and issues no error message. For further details, refer to SQL and Object Settings Pages.

From the Management Portal, **System Administration**, **Configuration**, **SQL and Object Settings**, **SQL** you can set this option (and other similar create, alter, and drop options) system-wide by selecting the **Ignore redundant DDL statements** check box.

The behavior of the predicate IF EXISTS takes priority over settings in the Management Portal and the configuration parameter file (CPF) which also govern DDL statements. These settings return SQLCODE 0 and suppress the error silently. When IF EXISTS is specified, the command returns SQLCODE 1 along with a message.

Journaling

If you specify the %NOJOURN keyword, then **DROP INDEX** suppresses journaling and disables transactions for the duration of the operation. To specify %NOJOURN, you must have %NOJOURN SQL administrative privileges, which you can set by using the GRANT command.

Table Name

If you specify the optional *table-name*, it must correspond to an existing table.

- If the specified *table-name* does not exist, InterSystems IRIS issues an SQLCODE -30 error and sets %msg to `Table 'SQLUser.tname' does not exist`.

- If the specified *table-name* exists but does not have an index named *index-name*, InterSystems IRIS issues an SQLCODE -333 error and sets %msg to `Attempt to DROP INDEX 'MyIndex' on table SQLUSER.TNAME failed - index not found.`

- If the specified *table-name* is a view, InterSystems IRIS issues an SQLCODE -333 error and sets %msg to `Attempt to DROP INDEX 'EmpSalaryIndex' on view SQLUSER.VNAME failed. Indices only supported for tables, not views.`

Arguments

IF EXISTS

An optional argument that suppresses the error if the command is executed on a nonexistent index. For further details, refer to the following section on nonexistent indexes.

index-name

The name of the index to be deleted. *index-name* is the SQL version of the name, which can include underscores and other punctuation. It is listed in the table's Management Portal SQL Catalog Details as the **SQL Map Name**.

ON table-name, ON TABLE table-name

An optional argument specifying the name of the table associated with the index. You can specify the *table-name* using either syntax: The first syntax uses the ON clause; the TABLE keyword is optional. The second syntax uses the qualified name syntax `schema-name.table-name.index-name`. A *table-name* can be qualified (schema.table), or unqualified (table). An unqualified table name takes the default schema name. If you omit the *table-name* entirely, InterSystems IRIS deletes the first index found that matches *index-name*, as described below.

Examples

The first example creates a table named Employee, which is used in all of the examples in this section.

The following example creates an index named "EmpSalaryIndex" and later removes it. Note that here DROP INDEX does not specify the table associated with the index; it assumes that "EmpSalaryIndex" is a unique index name in this namespace.

SQL

```
CREATE TABLE Employee (
    EMPNUM      INT NOT NULL,
    NAMELAST    CHAR(30) NOT NULL,
    NAMEFIRST   CHAR(30) NOT NULL,
    STARTDATE   TIMESTAMP,
    SALARY      MONEY,
    ACCRUEDVACATION   INT,
    ACCRUEDSICKLEAVE   INT,
    CONSTRAINT EMPLOYEEPK PRIMARY KEY (EMPNUM))
CREATE INDEX EmpSalaryIndex
    ON TABLE Employee
    (Namelast,Salary)
DROP INDEX EmpSalaryIndex
```

The following example specifies the table associated with the index to be dropped using an ON TABLE clause:

SQL

```
CREATE INDEX EmpVacaIndex
    ON TABLE Employee
    (NameLast,AccruedVacation)
DROP INDEX EmpVacaIndex ON TABLE Employee
```

The following example specifies the table associated with the index to be dropped using qualified name syntax:

SQL

```
CREATE INDEX EmpSickIndex
    ON TABLE Employee
    (NameLast,AccruedSickLeave)
DROP INDEX Employee.EmpSickIndex
```

The following command attempts to drop a nonexistent index. It generates an SQLCODE -333 error:

SQL

```
DROP INDEX PeopleIndex ON TABLE Employee
```

See Also

- CREATE INDEX

- Defining and Building Indexes

- SQLCODE error messages

DROP METHOD (SQL)

Deletes a method.

Synopsis

```
DROP METHOD [IF EXISTS] name [ FROM className ]
```

Description

The **DROP METHOD** command deletes a method. When you delete a method, InterSystems IRIS revokes it from all users and roles to whom it has been granted and removes it from the database.

In order to delete a method, you must have %DROP_METHOD administrative privilege, as specified by the GRANT command. If you are attempting to delete a method for a class with a defined owner, you must be logged in as the owner of the class. Otherwise, the system generates an SQLCODE -99 error (Privilege Violation).

You cannot drop a method if the class definition that contains that method definition is a deployed class. This operation fails with an SQLCODE -400 error with the %msg `Unable to execute DDL that modifies a deployed class: 'classname'`.

The following combinations of *name* and FROM *className* are supported. Note that the FROM clause specifies the class package name and method name, not the SQL names. In these examples, the system-wide default schema name is SQLUser, which corresponds to the User class package:

- `DROP METHOD BonusCalc FROM methBonusCalc`: drops the method SQLUser.BonusCalc().
- `DROP METHOD BonusCalc FROM User.methBonusCalc`: drops the method SQLUser.BonusCalc().
- `DROP METHOD Test.BonusCalc FROM methBonusCalc`: drops the method SQLUser.BonusCalc().
- `DROP METHOD BonusCalc FROM Employees.methBonusCalc`: drops the method Employees.BonusCalc().
- `DROP METHOD Test.BonusCalc FROM Employees.methBonusCalc`: drops the method Employees.BonusCalc().

If the specified method does not exist, **DROP METHOD** generates an SQLCODE -362 error. If the specified *className* does not exist, **DROP METHOD** generates an SQLCODE -360 error. If the specified method could refer to two or more methods, **DROP METHOD** generates an SQLCODE -361 error; you must specify a *className* to resolve this ambiguity.

If a method has been defined with the PROCEDURE characteristic keyword, you can determine if it exists in the current namespace by invoking the **$SYSTEM.SQL.Schema.ProcedureExists()** method. A method defined with the PROCEDURE keyword can be deleted either by **DROP METHOD** or **DROP PROCEDURE**.

You can also delete a method by removing the method from the class definition and then recompiling the class, or by deleting the entire class.

Arguments

IF EXISTS

An optional argument that suppresses the error if the command is executed on a nonexistent method.

name

The name of the method to be deleted. The name is an identifier. Do not specify the method's parameter parentheses. A *name* can be qualified (schema.name), or unqualified (name). An unqualified method name takes the default schema name, unless the FROM *className* clause is specified.

FROM className

If specified, the **FROM** *className* clause deletes the method from the given class. Note that you must specify the className of a method (methBonusCalc), not the SQL name (BonusCalc). If this clause is not specified, InterSystems IRIS searches all classes of the schema for the method, and deletes it. However, if no method of this name is found, or more than one method of this name is found, an error code is returned. If the deletion of the method results in an empty class, **DROP METHOD** deletes the class as well.

Examples

The following example attempts to delete mymeth from the class User.Employee. (Refer to **CREATE TABLE** for an example that creates class User.Employee.)

SQL

```
DROP METHOD mymeth FROM User.Employee
```

See Also

- CREATE METHOD
- SQLCODE error messages

DROP ML CONFIGURATION (SQL)

Deletes an ML configuration.

Synopsis

```
DROP ML CONFIGURATION ml-configuration-name
```

Arguments

ml-configuration-name	The name of the ML configuration to delete.

Description

The **DROP ML CONFIGURATION** command deletes an ML configuration and its corresponding class definition.

Conditions

- The ML configuration must exist in the current namespace. Attempting to delete a non-existent ML configuration generates an SQLCODE –30 error.

- You cannot delete the system default ML configuration. Attempting to do so results in a SQLCODE –189 error.

Required Security Privileges

Calling **DROP ML CONFIGURATION** requires %DROP_ML_CONFIGURATION privileges; otherwise, there is a SQLCODE –99 error (Privilege Violation). To assign %DROP_ML_CONFIGURATION privileges, use the GRANT command.

See Also

- ALTER ML CONFIGURATION, CREATE ML CONFIGURATION

DROP MODEL (SQL)

Deletes a model.

Synopsis

```
DROP MODEL model-name
```

Arguments

model-name	The name of the model to delete.

Description

The **DROP MODEL** command deletes a model and its corresponding class definition. It also purges any training runs and validation runs associated with the model.

Deleting a Non-Existent Model

The model must exist in the current namespace. Attempting to delete a non-existent model generates an SQLCODE —30 error.

Required Security Privileges

Calling **DROP MODEL** requires %MANAGE_MODEL privileges; otherwise, there is a SQLCODE –99 error (Privilege Violation). To assign %MANAGE_MODEL privileges, use the GRANT command.

See Also

- ALTER MODEL, CREATE MODEL

DROP PROCEDURE (SQL)

Deletes a procedure.

Synopsis

```
DROP PROCEDURE [IF EXISTS] procname [ FROM className ]
DROP PROC procname [ FROM className ]
```

Description

The **DROP PROCEDURE** command deletes a procedure in the current namespace. When you drop a procedure, InterSystems IRIS revokes it from all users and roles to whom it has been granted and removes it from the database.

In order to drop a procedure, you must have %DROP_PROCEDURE administrative privilege, as specified by the GRANT command. If you are attempting to delete a procedure for a class with a defined owner, you must be logged in as the owner of the class. Otherwise, the system generates an SQLCODE -99 error (Privilege Violation).

You cannot drop a procedure if the class definition that contains that procedure definition is a deployed class. This operation fails with an SQLCODE -400 error with the %msg `Unable to execute DDL that modifies a deployed class: 'classname'`.

The *procname* is not case-sensitive. You must specify *procname* without parameter parentheses; specifying parameter parentheses results in an SQLCODE -25 error.

The following combinations of *procname* and FROM *className* are supported. Note that the FROM clause specifies the class package name and procedure name, not the SQL names. In these examples, the system-wide default schema name is SQLUser, which corresponds to the User class package:

- `DROP PROCEDURE BonusCalc FROM procBonusCalc`: drops the procedure SQLUser.BonusCalc().

- `DROP PROCEDURE BonusCalc FROM User.procBonusCalc`: drops the procedure SQLUser.BonusCalc().

- `DROP PROCEDURE Test.BonusCalc FROM procBonusCalc`: drops the procedure SQLUser.BonusCalc().

- `DROP PROCEDURE BonusCalc FROM Employees.procBonusCalc`: drops the procedure Employees.BonusCalc().

- `DROP PROCEDURE Test.BonusCalc FROM Employees.procBonusCalc`: drops the procedure Employees.BonusCalc().

If the specified procedure does not exist, **DROP PROCEDURE** generates an SQLCODE -362 error. If the specified class does not exist, **DROP PROCEDURE** generates an SQLCODE -360 error. If the specified procedure could refer to two or more procedures, **DROP PROCEDURE** generates an SQLCODE -361 error; you must specify a *className* to resolve this ambiguity.

To determine if a specified *procname* exists in the current namespace, use the **$SYSTEM.SQL.Schema.ProcedureExists()** method. This method recognizes both procedures and methods defined with the PROCEDURE keyword. A method defined with the PROCEDURE keyword can be deleted using **DROP PROCEDURE**.

If you execute a **DROP PROCEDURE** for a procedure that is an ObjectScript class query procedure, InterSystems IRIS will also drop the methods related to the procedure, such as myprocExecute(), myprocGetInfo(), myprocFetch(), myprocFetchRows(), and myprocClose().

You can also delete a procedure by removing the stored procedure from the class definition and then recompiling the class, or by deleting the entire class.

Arguments

procname

The name of the procedure to be deleted. The name is an identifier. Do not specify the procedure's parameter parentheses. A *name* can be qualified (schema.name), or unqualified (name). An unqualified procedure name takes the default schema name, unless the FROM *className* clause is specified.

FROM className

If specified, the **FROM** *className* clause deletes the procedure from the given class. If this clause is not specified, Inter-Systems IRIS searches all classes of the schema for the procedure, and deletes it. However, if no procedure of this name is found, or more than one procedure of this name is found, an error code is returned. If the deletion of the procedure results in an empty class, **DROP PROCEDURE** deletes the class as well.

Examples

The following example attempts to delete myprocSP from the class User.Employee. (Refer to **CREATE TABLE** for an example that creates class User.Employee.)

SQL

```
DROP PROCEDURE myprocSP FROM User.Employee
```

See Also

- CREATE PROCEDURE

- SQLCODE error messages

DROP QUERY (SQL)

Deletes a query.

Synopsis

```
DROP QUERY [IF EXISTS] name [ FROM className ]
```

Description

The **DROP QUERY** command deletes a query. When you drop a query, InterSystems IRIS revokes it from all users and roles to whom it has been granted and removes it from the database.

In order to drop a query, you must have %DROP_QUERY administrative privilege, as specified by the GRANT command. If you are attempting to delete a query for a class with a defined owner, you must be logged in as the owner of the class. Otherwise, the system generates an SQLCODE -99 error (Privilege Violation).

You cannot drop a query if the class definition that contains that query definition is a deployed class. This operation fails with an SQLCODE -400 error with the %msg `Unable to execute DDL that modifies a deployed class: 'classname'.`

The following combinations of *name* and FROM *className* are supported. Note that the FROM clause specifies the class package name and query name, not the SQL names. In these examples, the system-wide default schema name is SQLUser, which corresponds to the User class package:

- `DROP QUERY BonusCalc FROM queryBonusCalc`: drops the query SQLUser.BonusCalc().
- `DROP QUERY BonusCalc FROM User.queryBonusCalc`: drops the query SQLUser.BonusCalc().
- `DROP QUERY Test.BonusCalc FROM queryBonusCalc`: drops the query SQLUser.BonusCalc().
- `DROP QUERY BonusCalc FROM Employees.queryBonusCalc`: drops the query Employees.BonusCalc().
- `DROP QUERY Test.BonusCalc FROM Employees.queryBonusCalc`: drops the query Employees.BonusCalc().

If the specified query does not exist, **DROP QUERY** generates an SQLCODE -362 error. If the specified class does not exist, **DROP QUERY** generates an SQLCODE -360 error. If the specified query could refer to two or more queries, **DROP QUERY** generates an SQLCODE -361 error; you must specify a *className* to resolve this ambiguity.

You can also delete a query by removing the query (projected as a stored procedure) from the class definition and then recompiling the class, or by deleting the entire class.

Arguments

IF EXISTS

An optional argument that suppresses the error if the command is executed on a nonexistent query.

name

The name of the query to be deleted. The name is an identifier. Do not specify the query's parameter parentheses. A *name* can be qualified (schema.name), or unqualified (name). An unqualified query name takes the system-wide default schema name, unless the FROM *className* clause is specified.

FROM className

If specified, the **FROM** *className* clause deletes the query from the given class. If this clause is not specified, InterSystems IRIS searches all classes of the schema for the query, and deletes it. However, if no query of this name is found, or more

than one query of this name is found, an error code is returned. If the deletion of the query results in an empty class, **DROP QUERY** deletes the class as well.

Examples

The following example attempts to delete myq from the class User.Employee. (Refer to **CREATE TABLE** for an example that creates class User.Employee.)

SQL

```
DROP QUERY myq FROM User.Employee
```

See Also

- CREATE QUERY
- SQLCODE error messages

DROP ROLE (SQL)

Deletes a role.

Synopsis

```
DROP ROLE [IF EXISTS] role-name
```

Description

The **DROP ROLE** statement deletes a role. When you drop a role, InterSystems IRIS revokes it from all users and roles to whom it has been granted and removes it from the database.

You can determine if a role exists by invoking the **$SYSTEM.SQL.Security.RoleExists()** method. If you attempt to drop a role that does not exist (or has already been dropped), **DROP ROLE** issues an SQLCODE -118 error.

Privileges

The **DROP ROLE** command is a privileged operation. Prior to using **DROP ROLE** in embedded SQL, it is necessary to fulfill at least one of the following requirements:

- You are the owner of the role.

- You are logged in with one of the following:

    - The %Admin_Secure administrative resource with USE permission

    - The %Admin_RoleEdit administrative resource with USE permission

    - Full security privileges on the system

- You were granted the role WITH ADMIN OPTION.

Failing to do so results in an SQLCODE –99 error (Privilege Violation).

Use the **$SYSTEM.Security.Login()** method to assign a user with appropriate privileges:

ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql(      )
```

You must have the `%Service_Login:Use` privilege to invoke the **$SYSTEM.Security.Login** method. For further information, see %SYSTEM.Security.

Arguments

IF EXISTS

An optional argument that suppresses the error if the command is executed on a nonexistent role.

role-name

The name of the role to be deleted. The name is an identifier. Role names are not case-sensitive.

Examples

The following example creates a role named BkUser and then deletes it:

SQL

```
CREATE ROLE BkName
DROP ROLE BkName
```

See Also

- SQL statements: CREATE ROLE, CREATE USER, DROP USER, GRANT, REVOKE, %CHECKPRIV

- SQL Users, Roles, and Privileges

- SQLCODE error messages

- ObjectScript: $ROLES and $USERNAME special variables

DROP SCHEMA (SQL)

Deletes the schema definition.

Synopsis

```
DROP SCHEMA [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

Arguments

Argument	Description
name	The name of the schema to be dropped. The name is an identifier.
IF EXISTS	*Optional* — Suppresses the error that arises if a schema with *name* does not exist.
CASCADE	*Optional* — Specifies that all objects with a schema are dropped, including tables, views, queries and methods projected as stored procedures, and user-defined aggregates.
RESTRICT	*Optional* — Specifies that the schema should only be dropped if nothing is defined within it. This option is assumed if CASCADE has not been specified.

Description

This command deletes a schema definition. The user that issues the command must either own the schema or have the %SQLSchemaAdmin resource in order to execute the operation.

If CASCADE is specified, all tables, views, queries and methods projected as stored procedures, and user-defined aggregates within the schema are dropped.

By default, the RESTRICT option is specified, but you may also specify it manually. When it is specified, the schema will only be dropped if nothing is defined within it. If DROP SCHEMA is specified without CASCADE and the schema is not empty, SQLCODE -475 is returned.

DROP SCHEMA provides an implicit %NOJOURN to suppress journaling and disable transactions while the operation is running. It also provides an implicit %DELDATA to delete data associated with the tables it drops when CASCADE has been specified.

If you run DROP SCHEMA on a schema that does not exist, SQLCODE -473 is returned.

See Also

- CREATE SCHEMA
- SQLCODE error messages listed in the *InterSystems IRIS Error Reference*

DROP TABLE (SQL)

Deletes a table and (optionally) its data.

Synopsis

```
DROP TABLE table [RESTRICT | CASCADE]
[%DELDATA | %NODELDATA]
```

Description

The **DROP TABLE** command deletes a table and its corresponding persistent class definition. If the table is the last item in its schema, deleting the table also deletes the schema and its corresponding persistent class package.

By default, **DROP TABLE** deletes both the table definition and the table's data (if any exists). The %NODELDATA keyword allows you to specify deletion of the table definition but not the table's data.

DROP TABLE deletes all indexes and triggers associated with the table.

In order to delete a table, the following conditions must be met:

- The table must exist in the current namespace. Attempting to delete a non-existent table generates an SQLCODE -30 error.

- The table definition must be modifiable. If the class that projects the table is defined without [DdlAllowed], attempting to delete the table generates an SQLCODE -300 error.

- The table must not be locked by another concurrent process. If the table is locked, **DROP TABLE** waits indefinitely for the lock to be released. If lock contention is a possibility, it is important that you LOCK the table IN EXCLUSIVE MODE before issuing a **DROP TABLE**.

- The table must either have no associated views or **DROP TABLE** must specify the CASCADE keyword. Attempting to delete a table with associated views without CASCADE generates an SQLCODE -321 error.

- You must have the necessary privileges to delete the table. Attempting to delete a table without the necessary privileges generates an SQLCODE -99 error.

- You *can* delete a table even if the corresponding class is defined as a deployed class.

- You cannot delete a table if the persistent class that projects the table has derived classes (subclasses). Attempting to delete a superclass that would leave a subclass orphaned generates an SQLCODE -300 error with a message: `Class 'MySuperClass' has derived classes and therefore cannot be dropped via DDL`.

You can use the **$SYSTEM.SQL.Schema.DropTable()** method to delete a table in the current namespace. You specify the SQL table name. Unlike **DROP TABLE**, this method can delete a table that was defined without [DdlAllowed]. The second argument specifies whether the table data should also be deleted; by default, data is not deleted.

ObjectScript

```
DO $SYSTEM.SQL.Schema.DropTable("Sample.MyTable",1,.SQLCODE,.%msg)
IF SQLCODE '= 0 {WRITE "SQLCODE ",SQLCODE," error: ",%msg}
```

You can use the **$SYSTEM.OBJ.Delete()** method to delete one or more tables in the current namespace. You must specify the persistent class name that projects the table (not the SQL table name). You can specify multiple class names using wildcards. The second argument specifies whether the table data should also be deleted; by default, data is not deleted.

Privileges

The **DROP TABLE** command is a privileged operation. The user must have %DROP_TABLE administrative privilege to execute **DROP TABLE**. Failing to do so results in an SQLCODE –99 error with the %msg `User does not have`

%DROP_TABLE privileges. You can use the GRANT command to assign %DROP_TABLE privileges, if you hold appropriate granting privileges.

It is not necessary for the user to have DELETE object privilege for the specified table, even when the **DROP TABLE** operation deletes both the table and the table data.

In embedded SQL, you can use the **$SYSTEM.Security.Login()** method to log in as a user with appropriate privileges:

ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql(      )
```

You must have the **%Service_Login:Use** privilege to invoke the **$SYSTEM.Security.Login** method. For further information, see %SYSTEM.Security.

DROP TABLE cannot be used on a table created by defining a persistent class, unless the table class definition includes [DdlAllowed]. Otherwise, the operation fails with an SQLCODE -300 error with the %msg DDL not enabled for class 'Schema.tablename'.

Existing Object Privileges

Deleting a table does not delete the object privileges for that table. For example, the privilege granted to a user to insert, update, or delete data on that table. This has the following two consequences:

- If a table is deleted, and then another table with the same name is created, users and roles will have the same privileges on the new table that they had on the old table.

- Once a table is deleted, it is not possible to revoke object privileges for that table.

For these reasons, it is generally recommended that you use the REVOKE command to revoke object privileges from a table before deleting the table.

Table Containing Data

By default, **DROP TABLE** deletes the table definition and deletes the table's data. This table data delete is an atomic operation; if **DROP TABLE** encounters data that cannot be deleted (for example, a row with a referential constraint) any data deletion already performed is automatically rolled back, with the result that no table data is deleted.

You can set the system-wide default for table data deletion using the **$SYSTEM.SQL.Util.SetOption()** method DDLDropTabDelData option. To determine the current setting, call **$SYSTEM.SQL.CurrentSettings()**, which displays the Does DDL DROP TABLE delete the table's data? setting.

The default is 1 ("Yes"). This is the recommended setting for this option. Set this option to 0 ("No") if you want **DROP TABLE** to not delete the table's data when it deletes the table definition.

The deletion of data can be overridden on a per-table basis. When deleting a table, you can specify **DROP TABLE** with the %NODELDATA option to prevent the automatic deletion of the table's data. If the system-wide default is set to not delete table data, you can delete data on a per-table basis by specifying **DROP TABLE** with the %DELDATA option.

In most circumstances **DROP TABLE** automatically deletes the table's data using a highly efficient kill extent operation. The following circumstances prevent the use of kill extent: the table has foreign keys that reference it; the class projecting the table is a subclass of a persistent class; the class does not use default storage; there is a ForEach = "row/object" trigger; there is a stream field that references a non-default stream field global location. If any of these apply, **DROP TABLE** deletes the table's data using a less-efficient delete record operation.

You can use the TRUNCATE TABLE command to delete the table's data without deleting the table definition.

Lock Applied

The **DROP TABLE** statement acquires an exclusive table-level lock on *table*. This prevents other processes from modifying the table definition or the table data while table deletion is in process. This table-level lock is sufficient for deleting both the table definition and the table data; **DROP TABLE** does not acquire a lock on each row of the table data. This lock is automatically released at the end of the **DROP TABLE** operation.

Foreign Key Constraints

By default, you cannot drop a table if any foreign key constraints are defined on another table that references the table you are attempting to drop. You must drop all referencing foreign key constraints before dropping the table they reference. Failing to delete these foreign key constraints before attempting a **DROP TABLE** operation results in an SQLCODE -320 error.

This default behavior is consistent with the RESTRICT keyword option. The CASCADE keyword option is not supported for foreign key constraints.

To change this default foreign key constraint behavior, refer to the COMPILEMODE=NOCHECK option of the SET OPTION command.

Associated Queries

Dropping a table automatically purges any related cached queries and purges query information as generated by %SYS.PTools.StatsSQL. Dropping a table automatically purges any SQL runtime statistics (SQL Stats) information for any related query.

Nonexistent Table

To determine if a specified table exists in the current namespace, use the **$SYSTEM.SQL.Schema.TableExists()** method.

By default, if you try to delete a nonexistent table, **DROP TABLE** issues an SQLCODE -30 error. This is the recommended setting. To determine the current setting, call **$SYSTEM.SQL.CurrentSettings()**, which displays a `Allow DDL DROP of non-existent table or view` setting. The default is 0 ("No"). If this option is set to 1 ("Yes"), **DROP TABLE** for a nonexistent table performs no operation and does not issue an error message.

From the Management Portal, **System Administration**, **Configuration**, **SQL and Object Settings**, **SQL** you can set this option (and other similar create, alter, and drop options) system-wide by selecting the **Ignore redundant DDL statements** check box.

Arguments

table

The name of the table to be deleted. The table name can be qualified (schema.table), or unqualified (table). An unqualified table name takes the default schema name. Schema search path values are not used.

RESTRICT, CASCADE

An optional argument. RESTRICT only allows a table with no dependent views or integrity constraints to be deleted. RESTRICT is the default if no keyword is specified. CASCADE allow a table with dependent views or integrity constraints to be deleted; any referencing views or integrity constraints will also be deleted as part of the table deletion. The CASCADE keyword option is not supported for foreign key constraints.

%DELDATA, %NODELDATA

These optional keywords specify whether to delete data associated with a table when deleting the table. The default is to delete table data.

Examples

The following example creates a table named SQLUser.MyEmployees and later deletes it. This example specifies that any data associated with this table *not* be deleted when the table is deleted:

SQL

```
CREATE TABLE SQLUser.MyEmployees (
NAMELAST       CHAR (30) NOT NULL,
NAMEFIRST      CHAR (30) NOT NULL,
STARTDATE      TIMESTAMP,
SALARY         MONEY)

DROP TABLE SQLUser.MyEmployees %NODELDATA
```

See Also

- ALTER TABLE, CREATE TABLE, TRUNCATE TABLE

- Defining Tables

- SQL and Object Settings Pages

- SQLCODE error messages

DROP TRIGGER (SQL)

Deletes a trigger.

Synopsis

```
DROP TRIGGER [IF EXISTS] name [ FROM table ]
```

Description

The **DROP TRIGGER** command deletes a trigger. If you wish to modify an existing trigger you must invoke **DROP TRIGGER** to delete the old version of the trigger before invoking **CREATE TRIGGER**.

Note: **DROP TABLE** drops all triggers associated with that table.

Privileges and Locking

The **DROP TRIGGER** command is a privileged operation. The user must have %DROP_TRIGGER administrative privilege to execute **DROP TRIGGER**. Failing to do so results in an SQLCODE –99 error with the %msg User 'name' does not have %DROP_TRIGGER privileges.

The user must have %ALTER privilege on the specified table. If the user is the Owner (creator) of the table, the user is automatically granted %ALTER privilege for that table. Otherwise, the user must be granted %ALTER privilege for the table. Failing to do so results in an SQLCODE –99 error with the %msg User 'name' does not have required %ALTER privilege needed to change the table definition for 'Schema.TableName'.

You can use the GRANT command to assign %DROP_TRIGGER and %ALTER privileges, if you hold appropriate granting privileges.

In embedded SQL, you can use the **$SYSTEM.Security.Login()** method to log in as a user with appropriate privileges:

ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql(        )
```

You must have the **%Service_Login:Use** privilege to invoke the **$SYSTEM.Security.Login** method. For further information, see %SYSTEM.Security.

- **DROP TRIGGER** cannot be used on a table projected from a persistent class, unless the table class definition includes [DdlAllowed]. Otherwise, the operation fails with an SQLCODE -300 error with the %msg DDL not enabled for class 'Schema.tablename'.

- **DROP TRIGGER** cannot be used on a table projected from a deployed persistent class. This operation fails with an SQLCODE -400 error with the %msg Unable to execute DDL that modifies a deployed class: 'classname'.

The **DROP TRIGGER** statement acquires a table-level lock on *table*. This prevents other processes from modifying the table's data. This lock is automatically released at the conclusion of the **DROP TRIGGER** operation.

FROM Clause

A trigger and its table must reside in the same schema. If the trigger name is unqualified, the trigger schema name defaults to the same schema as the table schema, as specified in the FROM clause. If the trigger name is unqualified, and there is no FROM clause, or the table name is also unqualified, the trigger schema defaults to the default schema name; schema search paths are not used. If both names are qualified, the trigger schema name must be the same as the table schema name.

A schema name mismatch results in an SQLCODE -366 error; this should only occur when both the trigger name and the table name are qualified and they specify different schema names.

In InterSystems SQL, a trigger name must be unique within its schema for a specific table. Thus it is possible to have more than one trigger in a schema with the same name. The optional FROM clause is used to determine which trigger to delete:

- If no FROM clause is specified, and InterSystems IRIS locates a unique trigger in the schema that matches the specified name, InterSystems IRIS deletes the trigger.

- If a FROM clause is specified, and InterSystems IRIS locates a unique trigger in the schema that matches both the specified name and the FROM table name, InterSystems IRIS deletes the trigger.

- If no FROM clause is specified, and InterSystems IRIS locates more than one trigger that matches the specified name, InterSystems IRIS issues an SQLCODE -365 error.

- If InterSystems IRIS locates no trigger that matches the specified name, either for the table specified in the FROM clause or, if there is no FROM clause, for any table in the schema, InterSystems IRIS issues an SQLCODE -363 error.

Arguments

IF EXISTS

An optional argument that suppresses the error if the command is executed on a nonexistent trigger.

name

The name of the trigger to be deleted. A trigger name may be qualified or unqualified; if qualified, its schema name must match the table's schema name.

FROM table

An optional argument that specifies the table the trigger is to be deleted from. If the FROM clause is specified, only the table is searched for the named trigger. If the FROM clause is not specified, the entire schema specified in *name* is searched for the named trigger.

Examples

The following example deletes a trigger named Trigger_1 associated with any table in the system-wide default schema. (The initial default schema is SQLUser):

SQL

```
DROP TRIGGER Trigger_1
```

The following example deletes a trigger named Trigger_2 associated with any table in the A schema.

SQL

```
DROP TRIGGER A.Trigger_2
```

The following example deletes a trigger named Trigger_3 associated with the Patient table in the system-wide default schema. If a trigger named Trigger_3 is found, but it is not associated with Patient, InterSystems IRIS issues an SQLCODE -363 error.

SQL

```
DROP TRIGGER Trigger_3 FROM Patient
```

The following examples all delete a trigger named Trigger_4 associated with the Patient table in the Test schema.

SQL

```
DROP TRIGGER Test.Trigger_4 FROM Patient
```

SQL

```
DROP TRIGGER Trigger_4 FROM Test.Patient
```

SQL

```
DROP TRIGGER Test.Trigger_4 FROM Test.Patient
```

See Also

- CREATE TRIGGER
- GRANT
- Using Triggers
- SQLCODE error messages

DROP USER (SQL)

Removes a user account.

Synopsis

```
DROP USER [IF EXISTS] user-name
```

Description

The **DROP USER** command removes a user account. This user account was created and the *user-name* specified using **CREATE USER**. If the specified *user-name* does not correspond to an existing user account, InterSystems IRIS issues an SQLCODE -118 error. You can determine if a user exists by invoking the **$SYSTEM.SQL.Security.UserExists()** method.

User names are not case-sensitive.

You can also delete a user by using the Management Portal. Select **System Administration**, **Security**, **Users** to list the existing users. On this table of user accounts you can click **Delete** for the user account you wish to delete.

Privileges

The **DROP USER** command is a privileged operation. Prior to using **DROP USER** in embedded SQL, you must be logged in as a user with one of the following:

- The %Admin_Secure administrative resource with USE permission

- The %Admin_UserEdit administrative resource with USE permission

- Full security privileges on the system

If you are not, the **DROP USER** command results in an SQLCODE -99 error (Privilege Violation).

Use the **$SYSTEM.Security.Login()** method to assign a user with appropriate privileges:

ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql(      )
```

You must have the **%Service_Login:Use** privilege to invoke the **$SYSTEM.Security.Login** method. For further information, see %SYSTEM.Security.

Arguments

user-name

An optional argument that suppresses the error if the command is executed on a nonexistent user.

Examples

You can drop PSMITH by issuing the statement:

SQL

```
DROP USER psmith
```

See Also

- SQL statements: CREATE USER, ALTER USER, GRANT, REVOKE, %CHECKPRIV

- [SQL Users, Roles, and Privileges](#)

- [SQLCODE error messages](#)

- ObjectScript: [$ROLES](#) and [$USERNAME](#) special variables

DROP VIEW (SQL)

Deletes a view.

Synopsis

```
DROP VIEW [IF EXISTS] view-name [CASCADE | RESTRICT]
```

Description

The **DROP VIEW** command removes a view, but does not remove the underlying tables or data.

A drop view operation can also be invoked using the **DropView()** method call:

```
$SYSTEM.SQL.Schema.DropView(viewname,SQLCODE,%msg)
```

Privileges

The **DROP VIEW** command is a privileged operation. Prior to using **DROP VIEW** it is necessary for your process to have either %DROP_VIEW administrative privilege or a DELETE object privilege for the specified view. Failing to do so results in an SQLCODE -99 error (Privilege Violation). You can determine if the current user has DELETE privilege by invoking the %CHECKPRIV command. You can determine if a specified user has DELETE privilege by invoking the **$SYSTEM.SQL.Security.CheckPrivilege()** method. You can use the **GRANT** command to assign %DROP_VIEW privileges, if you hold appropriate granting privileges.

In embedded SQL, you can use the **$SYSTEM.Security.Login()** method to log in as a user with appropriate privileges:

ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql(        )
```

You must have the **`%Service_Login:Use`** privilege to invoke the **$SYSTEM.Security.Login** method. For further information, see %SYSTEM.Security.

You *can* delete a view based on a table that is projected from a deployed persistent class.

Nonexistent View

To determine if a specified view exists in the current namespace, use the **$SYSTEM.SQL.Schema.ViewExists()** method.

By default, if you try to delete a nonexistent view, **DROP VIEW** issues an SQLCODE -30 error. To determine the current setting, call **$SYSTEM.SQL.CurrentSettings()**, which displays a `Allow DDL DROP of non-existent table or view` setting. The default is 0 ("No"). This is the recommended setting for this option. If set to 1 ("Yes") issuing a **DROP VIEW** or **DROP TABLE** for nonexistent views and tables performs no operation and issues no error message.

From the Management Portal, **System Administration**, **Configuration**, **SQL and Object Settings**, **SQL** you can set this option (and other similar create, alter, and drop options) system-wide by selecting the **Ignore redundant DDL statements** check box.

The behavior of the predicate IF EXISTS takes priority over settings in the Management Portal and the configuration parameter file (CPF) which also govern DDL statements. These settings return SQLCODE 0 and suppress the error silently. When IF EXISTS is specified, the command returns SQLCODE 1 along with a message.

VIEW Referenced by Other Views

If you try to delete a view referenced by other views in their queries, **DROP VIEW** issues an SQLCODE -321 error by default. This is the RESTRICT keyword behavior.

By specifying the CASCADE keyword, an attempt to delete a view referenced by other views in their queries succeeds. The **DROP VIEW** also deletes these other views. If InterSystems IRIS cannot perform all cascade view deletions (for example, due to an SQLCODE -300 error) no views are deleted.

Associated Queries

Dropping a view automatically purges any related cached queries and purges query information generated by %SYS.PTools.StatsSQL. Dropping a view automatically purges any SQL runtime statistics (SQL Stats) information for any related query.

Arguments

IF EXISTS

An optional argument that suppresses the error if the command is executed on a nonexistent view. For further details, refer to the following section on nonexistent tables.

view-name

The name of the view to be deleted. A view name can be qualified (schema.viewname), or unqualified (viewname). An unqualified view name takes the default schema name.

CASCADE, RESTRICT

An optional argument. Specify the CASCADE keyword to drop any other view that references *view-name*. Specify RESTRICT to issue an SQLCODE -321 error if there is another view that references *view-name*. The default is RESTRICT.

Examples

The following example creates a view named "CityAddressBook" and later deletes the view. Because it is specified with the RESTRICT keyword (the default), an SQLCODE -321 error is issued if the view is referenced by other views:

SQL

```
CREATE VIEW CityAddressBook AS
    SELECT Name,Home_Street FROM Sample.Person
    WHERE Home_City='Boston'
DROP VIEW CityAddressBook RESTRICT)
```

See Also

- ALTER VIEW, CREATE VIEW, GRANT

- Views

- SQL and Object Settings Pages

- SQLCODE error messages

EXPLAIN (SQL)

Returns the query plan(s) for a specified query.

Synopsis

```
EXPLAIN [ALT] [STAT]  [INTO :host-variable] query
```

Description

The **EXPLAIN** command returns the query plan for a specified query as an XML-tagged text string. This query plan is returned as a result set consisting of a single field named Plan.

The *query* must be a **SELECT**, **DELETE**, or **UPDATE** query. Specifying an **INSERT** query results in a result set with an empty Plan field. Specifying a *query* that begins with any other keyword results in an SQLCODE -51. You can use Show Plan to display a query plan for other queries, such as for INSERT queries which contain a SELECT clause.

The ALT and STAT keywords can be specified in any order. The INTO keyword must be specified after these keywords.

The user that issues the **EXPLAIN** command must have execute privileges for the %SYSTEM.QUERY_PLAN procedure.

The optional ALT keyword generates alternate query plans. All of the alternate query plans are returned in the same XML-tagged text string. The normalized *query* text (tagged as <sql>) is listed before each query plan.

The optional STAT keyword generates runtime performance statistics for each module in the query plan. The STAT keyword is only supported for SELECT queries. Runtime statistics are included in the same XML-tagged text string that contains the query plan. The following statistics are collected for each module:

- <ModuleName>: module name.

- <TimeSpent>: total execution time for the module, in seconds.

- <GlobalRefs>: a count of global references.

- <LinesOfCode>: a count of lines of code executed.

- <DiskWait>: disk wait time in seconds.

- <RowCount>: number of rows in result set.

- <ModuleCount>: number of times this module was executed.

- <Counter>: number of times this program was executed.

These statistics are returned within the text of the query plan(s) in the XML-tagged text string. Performance statistics for all modules in a query plan are returned before the associated query plan. Embedded SQL cannot generate or return runtime performance statistics; the STAT keyword is ignored and no error is issued.

The **EXPLAIN** command returns Show Plan results by invoking the **$SYSTEM,SQL.Explain()** method, then formatting the result set as a single field containing an XML-tagged text string.

The **EXPLAIN ALT** command returns the alternate show plans results by invoking the **$SYSTEM,SQL.Explain()** method with the all=1 qualifier, then formatting the result set as a single field containing an XML-tagged text string.

Note: This command is fully supported for use in Embedded SQL, Dynamic SQL, the SQL Shell, the Management Portal, and JDBC interfaces. It is not supported for ODBC interfaces.

Result Set XML Structure

The following is the structure of an XML-tagged text string for **EXPLAIN ALT STAT** *query*. Line breaks, indents, and comment notes are provided here for explanatory purposes:

```
<plans> /* tag included even if there is only one plan */
  <plan> /* the first query plan */
    <sql> /* the normalzed SELECT statement text */  </sql>
    <cost value="1147000"/>
      /* if STAT, include the following <stats> tags */
    <stats> <ModuleName>MAIN</ModuleName> /* XML-tagged list of stats (above) for MAIN module */ </stats>
    <stats> <ModuleName>FIRST</ModuleName> /* XML-tagged list of stats (above) for FIRST module */ </stats>
    <stats>  /* additional modules */ </stats>
  /* text of query plan */
  </plan>
  <plan> /* if ALT, same info for first alternate plan */
    ...
  </plan>
</plans>
```

The Explain() Method

You can return the same query plan information from ObjectScript using the **$SYSTEM.SQL.Explain**() method, as shown in the following example:

```
SET myquery=2
SET myquery(1)="SELECT Name,Age FROM Sample.Person WHERE Name %STARTSWITH 'Q' "
SET myquery(2)="ORDER BY Age"
SET status=$SYSTEM.SQL.Explain(.myquery,{"all":0},,.plan)
  IF status'=1 {WRITE "Explain() failed:" DO $System.Status.DisplayError(status) QUIT}
ZWRITE plan
```

Arguments

ALT

An optional argument that returns alternate query plans. The default is to return a single query plan.

STAT

(Dynamic SQL only): An optional argument that returns query plan runtime performance statistics. The default is to return query plan(s) without runtime statistics. This syntax is ignored for Embedded SQL.

INTO :host-variable

(Embedded SQL only): An optional output host variable into which the query plan(s) are placed. This syntax is ignored for Dynamic SQL.

query

A SELECT query.

Examples

This example returns the query plan as an XML string. It first returns the SQL query text, then the query plan:

```
EXPLAIN SELECT Name,DOB FROM Sample.Person WHERE Name [ 'Q'
```

This example returns the query plan and performance statistics as an XML string. It first returns the SQL query text, then the performance statistics (by module), then the query plan:

```
EXPLAIN STAT SELECT Name,DOB FROM Sample.Person WHERE Name [ 'Q'
```

This example returns alternate query plans as an XML string. It returns SQL query text before each query plan:

```
EXPLAIN ALT SELECT Name,DOB FROM Sample.Person WHERE Name [ 'Q'
```

This example returns a more complex query plan. Performance statistics appear both before and within the query plan:

```
EXPLAIN STAT SELECT p.Name AS Person, e.Name AS Employee
FROM Sample.Person AS p, Sample.Employee AS e
WHERE p.Name %STARTSWITH 'Q' GROUP BY e.Name ORDER BY p.Name
```

The following Embedded SQL example returns the query plan as an XML string. It first returns the SQL query text, then the query plan:

```
#sqlcompile select=Runtime
&sql(EXPLAIN INTO :qplan SELECT Name,DOB FROM Sample.Person WHERE Name [ 'Q')
WRITE qplan
```

The following Embedded SQL example returns alternative query plans as an XML string. It first returns the SQL query text, then the first query plan, then the SQL query text, then the second query plan, and so forth:

```
#sqlcompile select=Runtime
&sql(EXPLAIN ALT INTO :qplans SELECT Name,DOB FROM Sample.Person WHERE Name [ 'Q')
WRITE qplans
```

The following Embedded SQL example returns the query plan. The STAT keyword is ignored:

```
#sqlcompile select=Runtime
&sql(EXPLAIN STAT INTO :qplan SELECT Name,DOB FROM Sample.Person WHERE Name [ 'Q')
WRITE qplan
```

See Also

- SELECT

- JOIN

- Show Plan

- Runtime Performance Statistics

- Querying the Database

FETCH (SQL)

Repositions a cursor, and retrieves data from it.

Synopsis

```
FETCH cursor-name [INTO host-variable-list ]
```

Description

Within an embedded SQL application, a **FETCH** statement retrieves data from a cursor. The required sequence of actions is: **DECLARE**, **OPEN**, **FETCH**, **CLOSE**. Attempting a **FETCH** on a cursor that is not open results in an SQLCODE -102 error.

As an SQL statement, this is supported only from within embedded SQL. Equivalent operations are supported through ODBC using the ODBC API. For further details, refer to Embedded SQL.

An **INTO** clause can be specified as a clause of the **DECLARE** statement, as a clause of the **FETCH** statement, or both. The **INTO** clause allows data from the columns of a fetch to be placed into local host variables. Each host variable in the list, from left to right, is associated with the corresponding column in the cursor result set. The data type of each variable must either match or be a supported implicit conversion of the data type of the corresponding result set column. The number of variables must match the number of columns in the cursor select list.

The **FETCH** operation completes when the cursor advances to the end of the data. This sets SQLCODE=100 (No more data). It also sets the %ROWCOUNT variable to the number of fetched rows.

Note: The values returned by **INTO** clause host variables are only reliable while SQLCODE=0. If SQLCODE=100 (No more data) the host variable values should not be used.

The *cursor-name* is not namespace-specific. Changing the current namespace has no effect on use of a declared cursor. The only namespace consideration is that **FETCH** must occur in the namespace that contains the table(s) being queried.

%ROWID

When a **FETCH** retrieves a row of an updateable cursor, it sets %ROWID to the RowID value of the fetched row. An updateable cursor is one in which the top FROM clause contains exactly one element, either a table name or an updateable view name.

This setting of %ROWID for each row retrieved is subject to the following conditions:

- The **DECLARE cursorname CURSOR** and **OPEN cursorname** statements do not initialize %ROWID; the %ROWID value is unchanged from its prior value. The first successful **FETCH** sets %ROWID. Each subsequent **FETCH** that retrieves a row resets %ROWID to the current RowID. **FETCH** sets %ROWID if it retrieves a row of an updateable cursor. If the cursor is not updateable, %ROWID remains unchanged. If no rows matched the query selection criteria, **FETCH** does not change the prior the %ROWID value. Upon **CLOSE** or when **FETCH** issues an SQLCODE 100 (No Data, or No More Data), %ROWID contains the RowID of the last row retrieved.

- A cursor-based **SELECT** with a DISTINCT keyword or a GROUP BY clause does not set %ROWID. The %ROWID value is unchanged from its previous value (if any).

- A cursor-based **SELECT** that performs only aggregate operations does not set %ROWID. The %ROWID value is unchanged from its previous value (if any).

An Embedded SQL **SELECT** with no declared cursor does not set %ROWID. The %ROWID value is unchanged upon the completion of a simple **SELECT** statement.

FETCH for UPDATE or DELETE

You can use **FETCH** to retrieve a row for update or delete. The **UPDATE** or **DELETE** must specify the WHERE CURRENT OF clause. The **DECLARE** should specify the FOR UPDATE clause. The following example shows a cursor-based delete that deletes all selected rows:

ObjectScript

```
SET $NAMESPACE="Samples"
&sql(DECLARE MyCursor CURSOR FOR SELECT %ID,Status
     FROM Sample.Quality WHERE Status='Bad' FOR UPDATE)
&sql(OPEN MyCursor)
  IF SQLCODE<0 {WRITE "SQL Open Cursor Error:",SQLCODE," ",%msg  QUIT}
NEW %ROWCOUNT,%ROWID
FOR {&sql(FETCH MyCursor)  QUIT:SQLCODE'=0
     &sql(DELETE FROM Sample.Quality WHERE CURRENT OF MyCursor) }
WRITE !,"Number of rows updated=",%ROWCOUNT
&sql(CLOSE MyCursor)
    IF SQLCODE<0 {WRITE "SQL Close Cursor Error:",SQLCODE," ",%msg  QUIT}
```

Arguments

cursor-name

The name of a currently open cursor. The cursor name was specified in the DECLARE command. Cursor names are case-sensitive.

INTO host-variable-list

An optional argument that places data from the columns of a fetch into local variables. The *host-variable-list* specifies a host variable, or a comma-separated list of host variables, that are targets to contain data associated with the cursor. The **INTO** clause is optional. If it is not specified, the **FETCH** statement positions the cursor only.

Examples

The following Embedded SQL example shows **FETCH** invoked by an argumentless **FOR** loop retrieving data from a cursor named EmpCursor. The **INTO** clause is specified in the **DECLARE** statement:

ObjectScript

```
  &sql(DECLARE EmpCursor CURSOR FOR
     SELECT Name, Home_State
     INTO :name,:state FROM Sample.Employee
     WHERE Home_State %STARTSWITH 'M')
&sql(OPEN EmpCursor)
    IF SQLCODE<0 {WRITE "SQL Open Cursor Error:",SQLCODE," ",%msg  QUIT}
NEW %ROWCOUNT,%ROWID
FOR { &sql(FETCH EmpCursor)
     QUIT:SQLCODE'=0
     WRITE "count: ",%ROWCOUNT," RowID: ",%ROWID,!
     WRITE "  Name=",name," State=",state,! }
WRITE !,"Final Fetch SQLCODE: ",SQLCODE
&sql(CLOSE EmpCursor)
  IF SQLCODE<0 {WRITE "SQL Close Cursor Error:",SQLCODE," ",%msg  QUIT}
```

The following Embedded SQL example shows **FETCH** invoked by an argumentless **FOR** loop retrieving data from a cursor named EmpCursor. The **INTO** clause is specified as part of the **FETCH** statement:

ObjectScript

```
&sql(DECLARE EmpCursor CURSOR FOR
     SELECT Name,Home_State FROM Sample.Employee
     WHERE Home_State %STARTSWITH 'M')
&sql(OPEN EmpCursor)
     IF SQLCODE<0 {WRITE "SQL Open Cursor Error:",SQLCODE," ",%msg  QUIT}
FOR { &sql(FETCH EmpCursor INTO :name,:state)
     QUIT:SQLCODE'=0
     WRITE "count: ",%ROWCOUNT," RowID: ",%ROWID,!
     WRITE "  Name=",name," State=",state,! }
WRITE !,"Final Fetch SQLCODE: ",SQLCODE
&sql(CLOSE EmpCursor)
  IF SQLCODE<0 {WRITE "SQL Close Cursor Error:",SQLCODE," ",%msg  QUIT}
```

The following Embedded SQL example shows **FETCH** invoked using a **WHILE** loop:

ObjectScript

```
&sql(DECLARE C1 CURSOR FOR
     SELECT Name,Home_State INTO :name,:state FROM Sample.Person
     WHERE Home_State %STARTSWITH 'M')
&sql(OPEN C1)
     IF SQLCODE<0 {WRITE "SQL Open Cursor Error:",SQLCODE," ",%msg  QUIT}
&sql(FETCH C1)
WHILE (SQLCODE = 0) {
     WRITE "count: ",%ROWCOUNT," RowID: ",%ROWID,!
     WRITE "  Name=",name," State=",state,!
     &sql(FETCH C1) }
WRITE !,"Final Fetch SQLCODE: ",SQLCODE
&sql(CLOSE C1)
  IF SQLCODE<0 {WRITE "SQL Close Cursor Error:",SQLCODE," ",%msg  QUIT}
```

The following Embedded SQL example shows **FETCH** retrieving aggregate function values. %ROWID is not set:

ObjectScript

```
 &sql(DECLARE PersonCursor CURSOR FOR
     SELECT COUNT(*),AVG(Age) FROM Sample.Person )
&sql(OPEN PersonCursor)
     IF SQLCODE<0 {WRITE "SQL Open Cursor Error:",SQLCODE," ",%msg  QUIT}
NEW %ROWCOUNT
FOR { &sql(FETCH PersonCursor INTO :cnt,:avg)
     QUIT:SQLCODE'=0
     WRITE %ROWCOUNT," Num People=",cnt," Average Age=",avg,! }
WRITE !,"Final Fetch SQLCODE: ",SQLCODE
&sql(CLOSE PersonCursor)
  IF SQLCODE<0 {WRITE "SQL Close Cursor Error:",SQLCODE," ",%msg  QUIT}
```

The following Embedded SQL example shows **FETCH** retrieving DISTINCT values. %ROWID is not set:

ObjectScript

```
 &sql(DECLARE EmpCursor CURSOR FOR
     SELECT DISTINCT Home_State FROM Sample.Employee
     WHERE Home_State %STARTSWITH 'M'
     ORDER BY Home_State )
&sql(OPEN EmpCursor)
     IF SQLCODE<0 {WRITE "SQL Open Cursor Error:",SQLCODE," ",%msg  QUIT}
NEW %ROWCOUNT
FOR { &sql(FETCH EmpCursor INTO :state)
     QUIT:SQLCODE'=0
     WRITE %ROWCOUNT," State=",state,! }
WRITE !,"Final Fetch SQLCODE: ",SQLCODE
&sql(CLOSE EmpCursor)
  IF SQLCODE<0 {WRITE "SQL Close Cursor Error:",SQLCODE," ",%msg  QUIT}
```

The following Embedded SQL example shows that a cursor persists across namespaces. This cursor is declared in %SYS, opened and fetched in USER, and closed in SAMPLES. Note that the **OPEN** must be executed in the namespace that contains the table(s) being queried, and the **FETCH** must able to access the output host variables, which are namespace-specific:

```
  &sql(USE DATABASE %SYS)
  WRITE $ZNSPACE,!
&sql(DECLARE NSCursor CURSOR FOR SELECT Name INTO :name FROM Sample.Employee)
  &sql(USE DATABASE "USER")
  WRITE $ZNSPACE,!
&sql(OPEN NSCursor)
     IF SQLCODE<0 {WRITE "SQL Open Cursor Error:",SQLCODE," ",%msg  QUIT}
    NEW SQLCODE,%ROWCOUNT,%ROWID
FOR { &sql(FETCH NSCursor)
    QUIT:SQLCODE
    WRITE "Name=",name,! }
  &sql(USE DATABASE SAMPLES)
  WRITE $ZNSPACE,!
&sql(CLOSE NSCursor)
    IF SQLCODE<0 {WRITE "SQL Close Cursor Error:",SQLCODE," ",%msg  QUIT}
```

See Also

- CLOSE, DECLARE, OPEN

- SQL Cursors

- SQLCODE error messages

FREEZE PLANS (SQL)

Freezes one or more query plans.

Synopsis

```
FREEZE PLANS BY ID statement-hash
FREEZE PLANS BY TABLE table-name
FREEZE PLANS BY SCHEMA schema-name
FREEZE PLANS
```

Description

The **FREEZE PLANS** command freezes query plans. To unfreeze frozen query plans use the UNFREEZE PLANS command.

FREEZE PLANS can freeze query plans with the Plan State `Unfrozen`. It cannot freeze query plans with the Plan State `Unfrozen/Parallel`.

FREEZE PLANS provides four syntax forms for freezing query plans:

- A specified query plan: **FREEZE PLANS BY ID** *statement-hash*. The *statement-hash* value must be delimited by double quotation marks.

- All query plans for a table: **FREEZE PLANS BY TABLE** *table-name*. You can specify a table name or a view name. If a query plan references multiple tables and/or views, specifying any of these tables or views freezes the query plan.

- All query plans for all tables in a schema: **FREEZE PLANS BY SCHEMA** *schema-name*.

- All query plans for all tables in the current namespace: **FREEZE PLANS**.

This command issues SQLCODE 0 if one or more query plans are frozen; it issues SQLCODE 100 if no query plans are frozen. The Rows Affected (%ROWCOUNT) indicates the number of query plans frozen.

Other Interfaces

You can use the following $SYSTEM.SQL.Statement methods to freeze a single query plan or multiple query plans: **FreezeStatement()** for a single plan; **FreezeRelation()** for all plans for a relation (a table or view referenced in the query plan); **FreezeSchema()** for all plans for a schema; **FreezeAll()** for all plans in the current namespace. There are corresponding Unfreeze methods.

You can use the Management Portal, to freeze a query plan as described in Frozen Plans Interface.

Arguments

statement-hash

The internal hash representation of the SQL Statement definition for a query plan, enclosed in quotation marks. Occasionally, what appear to be identical SQL statements may have different statement hash entries. Any difference in settings/options that require different code generation of the SQL statement result in a different statement hash. This may occur with different client versions or different platforms that support different internal optimizations. Refer to SQL Statement Details.

table-name

The name of an existing table or view. A *table-name* can be qualified (schema.table), or unqualified (table). An unqualified table name takes the default schema name.

schema-name

The name of an existing schema. This command freezes all query plans for all tables in the specified schema.

Security and Privileges

The **FREEZE PLANS** command is a privileged operation that required the user to have %Development:USE permission. Such permissions can be granted through the Management Portal. Executing a **FREEZE PLANS** command without this privileges will result in a SQLCODE -99 error and the command will fail. There are two exceptions:

- The command is executed via Embedded SQL, which does not perform privilege checks.

- The user explicitly specifies not privilege checking by, for example, calling either **%Prepare()** with the checkPriv argument set to 0 or **%ExecDirectNoPriv()** on a %SQL.Statement.

See Also

- UNFREEZE PLANS command
- Frozen Plans
- Analyze SQL Statements and Statistics

GRANT (SQL)

Grants privileges to a user or role.

Synopsis

```
GRANT admin-privilege TO grantee [WITH ADMIN OPTION]
GRANT role TO grantee [WITH ADMIN OPTION]

GRANT role TO grantee [WITH ADMIN OPTION]

GRANT object-privilege ON object-list TO grantee [WITH GRANT OPTION]

GRANT SELECT ON CUBE[S] object-list TO grantee [WITH GRANT OPTION]
GRANT column-privilege (column-list) ON table TO grantee  [WITH GRANT OPTION]
```

Description

The **GRANT** command gives privileges to do specified tasks on specified tables, views, columns, or other entities to one or more specified users or roles. You can do the following basic operations:

- Grant a privilege to a user.

- Grant a privilege to a role.

- Grant a role to a user.

- Grant a role to a role, creating a hierarchy of roles.

If you grant a privilege to a user, the user can immediately exercise the privilege. If you grant a privilege to a role, users who have been granted the role can immediately exercise the privilege. If you revoke a privilege, the user immediately loses the privilege. A privilege is effectively granted to a user only once. Multiple users can grant the same privilege to a user multiple times, but a single **REVOKE** removes the privilege.

Privileges are granted on a per-namespace basis.

SQL privileges are only enforced through ODBC, JDBC, and Dynamic SQL (%SQL.Statement).

Because **GRANT** prepares and executes quickly, and is generally run only once, InterSystems IRIS does not create a cached query for **GRANT** in ODBC, JDBC, or Dynamic SQL. The expansion of * is performed when the **GRANT** command is executed.

GRANT admin-privilege

SQL administrative (admin) privileges apply to users or roles. Any privilege that is not tied to any particular object (and thus is a general right for that user or role) is considered an admin privilege. These privileges are granted on a per-namespace basis for the current namespace.

The %DB_OBJECT_DEFINITION privilege grants all 16 of the data definition privileges. It does not grant %BUILD_INDEX, %NOCHECK, %NOINDEX, %NOLOCK, and %NOTRIGGER privileges, which must be granted explicitly.

The %BUILD_INDEX privilege grants use of the BUILD INDEX command. The %NOCHECK, %NOINDEX, %NOLOCK, and %NOTRIGGER privileges grant use of these options in the *restriction* clause of an INSERT, UPDATE, INSERT OR UPDATE, or DELETE statement. They have no effect on the use of the %NOINDEX keyword as a preface to a predicate condition. Because TRUNCATE TABLE performs a delete of all of the rows from a table with %NOTRIGGER behavior, you must have %NOTRIGGER privilege in order to run **TRUNCATE TABLE**. You must have the appropriate %NOCHECK, %NOINDEX, %NOLOCK, or %NOTRIGGER privilege to use that *restriction* when preparing an **INSERT**, **UPDATE**, **INSERT OR UPDATE**, or **DELETE** statement.

If the specified admin privilege is not a valid privilege name (for example, due to a spelling error), InterSystems IRIS completes successfully, issuing an SQLCODE 100 (reached end of data); InterSystems IRIS does not check if the specified user (or role) exists. If the specified admin privilege is valid, but the specified user (or role) does not exist, InterSystems IRIS issues an SQLCODE -118 error.

GRANT role

This form of **GRANT** assigns a user to a specified role. You can also assign a role to another role. If the specified role that receives the assignment does not exist, InterSystems IRIS issues an SQLCODE 100 (reached end of data). If the specified user (or role) that is assigned to a role does not exist, InterSystems IRIS issues an SQLCODE -118 error. If you are not the SuperUser, and you are attempting to grant a role that you don't own and don't have ADMIN OPTION for, InterSystems IRIS issues an SQLCODE -112 error.

Roles are created using the CREATE ROLE statement. If the role name is a delimited identifier, you must enclose it in quotation marks when assigning to it.

Roles can be granted or revoked via either the SQL **GRANT** and **REVOKE** commands, or via InterSystems IRIS System Security:

- Go to the Management Portal, select **System Administration**, **Security**, **Users** to display the current users. Select the name of the desired user to display edit options for that user, then select the **Roles** tab to assign (or unassign) the user to one or more roles.

- Go to the Management Portal, select **System Administration**, **Security**, **Roles** to display the current roles. Select the name of the desired role to display edit options for that role, then select the **Assigned To** tab to assign (or unassign) the role to one or more roles.

GRANT object-privilege

Object privileges give a user or role some right to a particular object. You grant an *object-privilege* ON an *object-list* TO a *grantee*. An *object-list* can specify one or more tables, views, stored procedures, or cubes in the current namespace. By using comma-separated lists, a single **GRANT** statement can grant multiple object privileges on multiple objects to multiple users and/or roles.

The following are the available *object-privilege* values:

- The %ALTER and DELETE privileges grant access to table or view definitions.

- The SELECT, INSERT, UPDATE, DELETE, and REFERENCES privileges grant access to table data.

- The EXECUTE privilege grants access to stored procedures. This privilege is required to execute a stored procedure or to call a user-defined SQL function in a query. For example, `SELECT Field1,MyFunc() FROM SQLUser.MyTable` requires SELECT privilege on SQLUser.MyTable and EXECUTE privilege on the SQLUser.MyFunc procedure.

- The ALL PRIVILEGES privilege grants all table and view privileges; it does not grant the EXECUTE privilege.

You can use the asterisk (*) wildcard as the *object-list* value to grant the *object-privilege* to all of the objects in the current namespace. For example, `GRANT SELECT ON * TO Deborah` grants this user SELECT privilege for all tables and views. `GRANT EXECUTE ON * TO Deborah` grants this user EXECUTE privilege for all non-hidden Stored Procedures.

You can use SCHEMA *schema-name* as the *object-list* value to grant the *object-privilege* to all of the tables, views, and stored procedures in the named schema, in the current namespace. The following example grants this user SELECT privilege for all objects in the Sample schema.

```
GRANT SELECT ON SCHEMA Sample TO Deborah
```

This includes all objects that will be defined in this schema in the future. You can specify multiple schemas as a comma-separated list, as in the following example, which grants SELECT privilege for all objects in both the Sample and the Cinema schemas.

```
GRANT SELECT ON SCHEMA Sample,Cinema TO Deborah
```

Cubes are SQL identifiers that are not qualified by a schema name. To specify a cubes *object-list*, you must specify the CUBE (or CUBES) keyword. You can only grant SELECT privilege to a cube.

The following example demonstrates the granting of the SELECT and UPDATE privileges to a specific user for a specific table:

ObjectScript

```
    DO $SYSTEM.Security.Login("_SYSTEM","SYS")
CreateUser
    SET x=$SYSTEM.SQL.Security.UserExists("DeborahTest")
    IF x=0 {&sql(CREATE USER DeborahTest IDENTIFY BY birdpw)
            IF SQLCODE '= 0 {WRITE "CREATE USER error: ",SQLCODE,!
                             QUIT}
          }
    ELSE {WRITE "User DeborahTest exists, not changing privileges",!
         QUIT }
GrantPrivsToUser
    &sql(GRANT SELECT,UPDATE ON SQLUSER.T1 TO DeborahTest)
    WRITE !,"GRANT error code: ",SQLCODE
DropUser
    &sql(DROP USER DeborahTest)
      IF SQLCODE '= 0 {WRITE "DROP USER error: ",SQLCODE,!}
```

Privileges can only be granted explicitly to a table, view, or stored procedure that already exists. If the specified object does not exist, InterSystems IRIS issues an SQLCODE -30 error. You can, however, grant privileges to a schema, which grant privileges both to all existing objects in that schema and to all future objects in that schema that did not exist when the privilege was granted.

If the owner of a table is _PUBLIC, users do not need to be granted object privileges to access the table.

If the specified user does not exist, InterSystems IRIS issues an SQLCODE -118 error. If the specified object privilege has already been granted, InterSystems IRIS issues an SQLCODE 100 (reached end of data).

Object privileges can be granted or revoked by any of the following:

- The **GRANT** and **REVOKE** commands.

- The **$SYSTEM.SQL.Security.GrantPrivilege()** and **$SYSTEM.SQL.Security.RevokePrivilege()** methods. These methods return a %Status value and set the SQLCODE variable. As with any method or function, always test the returned value first:

  - If %Status=1 and SQLCODE=0: a privilege was granted or revoked.

  - If %Status=1 and SQLCODE=100: no privilege was granted or revoked because it already has been granted or revoked.

  - If %Status is not 1 and SQLCODE is not set and may be undefined: no privilege was granted or revoked due to a method error. The %Status contains an SQLCODE indicating the type of failure: *ObjPriv*: SQLCODE -60 for an invalid privilege; *ObjList*: an ObjList object of the specified Object Type does not exist: SQLCODE -30, -187, -428, or -473; *Type*: SQLCODE -400 Object Type of TABLE, VIEW, CUBES, SCHEMA, or STORED PROCE-DURES expected; *User*: SQLCODE -118 Unknown or non-unique user or role.

- Via InterSystems IRIS System Security. Go to the Management Portal, select **System Administration**, **Security**, **Users** (or **System Administration**, **Security**, **Roles**) select the name of the desired user or role, then select the **SQL Tables** or **SQL Views** tab. Select the desired **Namespace** from the drop-down list. Then select the **Add Tables** or **Add Views** button. In the displayed window, choose a schema, select one or more tables, and assign privileges.

You can determine if the current user has a specified object privilege by invoking the %CHECKPRIV command. You can determine if a specified user has a specified table-level object privilege by invoking the **$SYSTEM.SQL.Security.CheckPrivilege()** method, as shown in the following example:

ObjectScript

```
WRITE "SELECT privilege? ",$SYSTEM.SQL.Security.CheckPrivilege("DeborahTest","1,SQLUSER.TestT1","s"),!

WRITE "UPDATE privilege? ",$SYSTEM.SQL.Security.CheckPrivilege("DeborahTest","1,SQLUSER.TestT1","u"),!

WRITE "DELETE privilege? ",$SYSTEM.SQL.Security.CheckPrivilege("DeborahTest","1,SQLUSER.TestT1","d"),!
```

Object Owner Privileges

The owner of a table, view, or procedure always has all SQL privileges implicitly on the SQL object. The owner of the object has privileges on the object in all namespaces to which the object is mapped.

GRANT column-privilege

Column privileges give a user or role a specified privilege to a specified list of columns on a specified table or view. This permits you to allow access to some table columns and not to other columns of the same table. This gives more specific access control than the GRANT object-privilege option, which defines privileges for an entire table or view. When granting privileges to a grantee, you should grant either table-level privilege or column-level privileges for a table, but not both. The SELECT, INSERT, UPDATE, and REFERENCES privileges can be used to grant access to data in individual columns.

A user having a SELECT, INSERT, UPDATE, or REFERENCES *object-privilege* on a table WITH GRANT OPTION can grant to other users a *column-privilege* of the same type for columns of that table.

You can specify a single column, or a comma-separated list of columns. The *column-list* must be enclosed in parentheses. Column names can be specified in any order, and duplication is permitted. Granting a column privilege to a column that already has that privilege has no effect.

The following example grants the UPDATE privilege for two columns:

SQL

```
GRANT UPDATE(Name,FavoriteColors) ON Sample.Person TO Deborah
```

You can grant column privileges on a table or a view. You can grant column privileges to any type of *grantee*, including a list of users, a list of roles, *, and _PUBLIC. However, you cannot use the asterisk (*) wildcard for privileges, field names, or table names.

If a user inserts a new record into a table, data is inserted into only those fields for which column privileges have been granted. All other data columns are set to either the defined column default value, or to NULL if there is no defined default value. You cannot grant column-level INSERT or UPDATE privileges to the RowID and Identity columns. Upon INSERT, InterSystems SQL automatically provides a RowID and (if needed) an Identity column value.

Column-level privileges can be granted or revoked via either the SQL **GRANT** and **REVOKE** commands, or via InterSystems IRIS System Security. Go to the Management Portal, select **System Administration**, **Security**, **Users** (or **System Administration**, **Security**, **Roles**), select the name of the desired user or role, then select the **SQL Tables** or **SQL Views** tab. Select the desired **Namespace** from the drop-down list. Then select the **Add Columns** button. In the displayed window, choose a schema, choose a table, select one or more columns, and assign privileges.

Granting Multiple Privileges

You can use a single **GRANT** statement to specify the following combinations of privileges:

- One or more roles.

- One or more table-level privileges and one or more column-level privileges. To specify multiple table-level and column-level privileges, the privilege must immediately precede a *column-list* to grant a column-level privilege. Otherwise, it grants a table-level privilege.

- One or more admin-privileges. You cannot include admin-privileges and role names or object privileges in the same **GRANT** statement. Attempting to do so results in an SQLCODE -1 error.

The following example grants Deborah table-level SELECT and UPDATE privileges, and column-level INSERT privileges:

SQL

```
GRANT SELECT,UPDATE,INSERT(Name,FavoriteColors) ON Sample.Person TO Deborah
```

The following example grants Deborah column-level SELECT, INSERT, and UPDATE privileges:

SQL

```
GRANT SELECT(Name,FavoriteColors),INSERT(Name,FavoriteColors),UPDATE(FavoriteColors) ON Sample.Person
 TO Deborah
```

The WITH GRANT OPTION Clause

The owner of an object automatically holds all privileges on that object. The **GRANT** statement's TO clause specifies the users or roles to whom to access is being granted. After using the TO option to specify the grantee, you may optionally specify the WITH GRANT OPTION keyword clause to allow the grantee(s) to also be able to grant the same privileges to other users. You can use the WITH GRANT OPTION keyword clause with object privileges or column privileges. The **REVOKE** command with CASCADE can be used to undo this cascading series of granted privileges.

For instance, you can give the user Chris %ALTER, SELECT, and INSERT privileges on the EMPLOYEES table with the following command:

SQL

```
GRANT %ALTER, SELECT, INSERT
     ON EMPLOYEES
     TO Chris
```

To also give Chris the ability to give these privileges to other users, the GRANT command includes the WITH GRANT OPTION clause:

SQL

```
GRANT %ALTER, SELECT, INSERT
     ON EMPLOYEES
     TO Chris WITH GRANT OPTION
```

You can find out the results of a GRANT statement using the **%SQLCatalogPriv.SQLUsers()** method call.

Granting privileges to a schema WITH GRANT OPTION allow the grantee(s) to be able to grant the same schema privileges to other users. However, it does not allow the grantee to grant a privilege on a specified object within that schema, unless the user has been explicitly granted the privilege on that particular object WITH GRANT OPTION. This is shown in the following example:

- UserA and UserB start with no privileges.

- You grant UserA SELECT privilege on schema Sample WITH GRANT OPTION.

- UserA can grant SELECT privilege on schema Sample to UserB.

- UserA *cannot* grant SELECT privilege on table Sample.Person to UserB.

The WITH ADMIN OPTION Clause

The WITH ADMIN OPTION clause grants the *grantee* the right to grant the same privileges it received to others. To grant a system privilege, you must have been granted the system privilege WITH ADMIN OPTION.

You may grant a role if either the role has been granted to you WITH ADMIN OPTION, or if you have the %Admin_Secure:"U" resource.

A grant WITH ADMIN OPTION supersedes a previous grant of the same privilege(s) without this option. Thus, if you grant a user a privilege without WITH ADMIN OPTION, and then grant the same privilege to the user WITH ADMIN OPTION, the user has the WITH ADMIN OPTION rights. However, a grant without the WITH ADMIN OPTION *does not* supersede a previous grant of the same privilege(s) with this option. To remove WITH ADMIN OPTION rights from a privilege, you must revoke the privilege and then re-grant the privilege without this clause.

Exporting Privileges

You can export privileges using the **$SYSTEM.SQL.Schema.ExportDDL()** method. When you specify a table in this method, InterSystems IRIS exports both all table-level privileges and all column-level privileges granted for that table. For further details, refer to the *InterSystems Class Reference*.

InterSystems IRIS Security

Before using **GRANT** in embedded SQL, it is necessary to be logged in as a user with appropriate privileges. Failing to do so results in an SQLCODE -99 error (Privilege Violation). Use the **$SYSTEM.Security.Login()** method to assign a user with appropriate privileges:

ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql(        )
```

You must have the `%Service_Login:Use` privilege to invoke the **$SYSTEM.Security.Login** method. For further information, refer to %SYSTEM.Security in the *InterSystems Class Reference*.

Enforcement of Privileges

SQL privileges are only enforced through ODBC, JDBC, and Dynamic SQL (%SQL.Statement).

The enforcement of privileges system-wide depends upon the setting of the **$SYSTEM.SQL.Util.SetOption("SQLSecurity")** method call. To determine the current setting, call **$SYSTEM.SQL.CurrentSettings()**, which displays a SQL Security ON: setting.

The default is 1 (Yes): a user can only perform actions on tables and views for which that user has been granted privilege. This is the recommended setting for this option. If this option is set to 0 (No), SQL Security is disabled for any new process started after changing this setting. This means privilege-based table/view security is suppressed. You can create a table without specifying a user. In this case, the Management Portal assigns "_SYSTEM" as user, and embedded SQL assigns "" (the empty string) as user. Any user can perform actions on a table or view even if that user has no privileges to do so.

Arguments

grantee

A comma-separated list of one or more users or roles. Valid values are a list of users, a list of roles, "*", or _PUBLIC. The asterisk (*) specifies all currently defined users who do not have the %All role. The _PUBLIC keyword specifies all currently defined and yet-to-be-defined users.

admin-privilege

An administrative-level privilege or a comma-separated list of administrative-level privileges being granted. The list may consist of one or more of the following in any order:

%CREATE_METHOD, %DROP_METHOD, %CREATE_FUNCTION, %DROP_FUNCTION, %CREATE_PROCEDURE, %DROP_PROCEDURE, %CREATE_QUERY, %DROP_QUERY, %CREATE_TABLE, %ALTER_TABLE, %DROP_TABLE, %CREATE_VIEW, %ALTER_VIEW, %DROP_VIEW, %CREATE_TRIGGER, %DROP_TRIGGER

%DB_OBJECT_DEFINITION, which grants all 16 of the above privileges.

%NOCHECK, %NOINDEX, %NOLOCK, %NOTRIGGER privileges for INSERT, UPDATE, and DELETE operations.

%BUILD_INDEX which grants privileges for the BUILD INDEX command.

role

A role or comma-separated list of roles whose privileges are being granted.

object-privilege

A basic-level privilege or comma-separated list of basic-level privileges being granted. The list may consist of one or more of the following: %ALTER, DELETE, SELECT, INSERT, UPDATE, EXECUTE, and REFERENCES. You can confer all table and view privileges using either "ALL [PRIVILEGES]" or "*" as the argument value. Note that you can only grant SELECT privilege to CUBES.

object-list

A comma-separated list of one or more tables, views, stored procedures, or cubes for which the *object-privilege*(s) are being granted. You can use the SCHEMA keyword to specify granting the *object-privilege* to all objects in the specified schema. You can use "*" to specify granting the *object-privilege* to all tables, or to all non-hidden Stored Procedures, in the current namespace. Note that a cubes *object-list* requires the CUBE (or CUBES) keyword, and can only be granted SELECT privilege.

column-privilege

A basic-level privilege being granted to one or more listed columns. Available options are SELECT, INSERT, UPDATE, and REFERENCES.

column-list

A list of one or more column names, separated by commas and enclosed in parentheses.

table

The name of the table or view that contains the *column-list* columns.

Examples

The following example creates a user, creates a role, and then assigns the role to the user. If the user or role already exists, it issues SQLCODE -118 error. If the assignment of the privilege or the role has already been done, no error is issued (SQLCODE = 0).

ObjectScript

```
   DO $SYSTEM.Security.Login("_SYSTEM","SYS")
CreateUser
   SET x=$SYSTEM.SQL.Security.UserExists("MarthaTest")
   IF x=0 {&sql(CREATE USER MarthaTest IDENTIFY BY birdpw)
          IF SQLCODE '= 0 {WRITE "CREATE USER error: ",SQLCODE,!
                           QUIT}
        }
   ELSE {WRITE "User MarthaTest exists, not changing its roles",!
        QUIT }
CreateRoleAndGrant
   &sql(CREATE ROLE workerbee)
   WRITE !,"CREATE ROLE error code: ",SQLCODE
   &sql(GRANT %CREATE_TABLE TO workerbee)
   WRITE !,"GRANT privilege error code: ",SQLCODE
   &sql(GRANT workerbee TO MarthaTest)
   WRITE !,"GRANT role error code: ",SQLCODE
```

The following example shows the assignment of multiple privileges. It creates a user and creates two roles. A single **GRANT** statement assigns these roles and a list of admin-privileges to the user. If the user or a role already exists, it issues SQLCODE -118 error. If the assignment of a privilege or a role has already been done, no error is issued (SQLCODE = 0).

ObjectScript

```
   DO $SYSTEM.Security.Login("_SYSTEM","SYS")
CreateUser
   SET x=$SYSTEM.SQL.Security.UserExists("NoahTest")
   IF x=0 {&sql(CREATE USER NoahTest IDENTIFY BY birdpw)
          IF SQLCODE '= 0 {WRITE "CREATE USER error: ",SQLCODE,!
                           QUIT}
          }
   ELSE {WRITE "User NoahTest exists, not changing its roles",!
         QUIT }
Create2RolesAndGrant
   &sql(CREATE ROLE workerbee)
   WRITE !,"CREATE ROLE 1 error code: ",SQLCODE
   &sql(CREATE ROLE drone)
   WRITE !,"CREATE ROLE 2 error code: ",SQLCODE
   &sql(GRANT workerbee,drone,%CREATE_TABLE,%DROP_TABLE TO NoahTest)
   WRITE !,"GRANT roles & privileges error code: ",SQLCODE
```

The following example grants all seven basic privileges ON all tables in the current namespace TO all currently defined users who do not have the %All role:

ObjectScript

```
   DO $SYSTEM.Security.Login("_SYSTEM","SYS")
   &sql(GRANT * ON * TO *)
```

See Also

- %CHECKPRIV REVOKE

- SELECT INSERT DELETE UPDATE

- CREATE USER CREATE ROLE

- SQL Users, Roles, and Privileges

- CREATE FUNCTION CREATE METHOD CREATE PROCEDURE CREATE QUERY

- CREATE TABLE CREATE VIEW CREATE TRIGGER

- SQLCODE error messages

- ObjectScript: $ROLES and $USERNAME special variables

INSERT (SQL)

Adds new rows to a table.

Synopsis

Single Row Inserts

```
INSERT INTO table (column, column2, ...) VALUES (value, value2, ...)

INSERT INTO table VALUES (value, value2, ...)

INSERT INTO table SET column = value, column2 = value2, ...

INSERT INTO table DEFAULT VALUES

INSERT INTO table VALUES :array()
```

Multi-Row Inserts

```
INSERT INTO table query

INSERT INTO table (column, column2, ...) query
```

Insert Options

```
INSERT table ...

INSERT %keyword [INTO] table ...
```

Description

The **INSERT** command inserts a row into a table or uses the results of a **SELECT** query to insert multiple rows into a table. This command inserts data for all specified columns and defaults unspecified column values to either NULL or to the defined default value. It sets the %ROWCOUNT variable to the number of inserted rows.

If the row being inserted already exists (for example, it fails a UNIQUE check), **INSERT** generates an error. To update existing rows in these cases, use INSERT OR UPDATE.

Single Row Inserts

- **INSERT INTO** *table* **(***column***,** *column2***, ...) VALUES (***value***,** *value2***, ...)** inserts a row of values into the specified columns of a table. The values in the VALUES clause must correspond positionally with the column names in the column list.

  By default, an **INSERT** is an all-or-nothing event: either a row is inserted completely or not at all. InterSystems IRIS® returns a status variable SQLCODE, indicating the success or failure of the **INSERT**. To insert a row into a table, the insert must meet all requirements described in the *table*, *column*, and *value* arguments.

  This statement inserts a new row into the Sample.Records table, setting the value of the StatusDate column to '05/12/22' and the value of the Status column to 'Purged'.

  SQL

  ```
  INSERT INTO Sample.Records (StatusDate,Status) VALUES ('05/12/22','Purged')
  ```

  Example: Insert Rows into Table Using Specified Values

- **INSERT INTO** *table* **VALUES (***value***,** *value2***, ...)** inserts the table row of values in column number order. The data values must correspond positionally to the defined column list. You must specify a value for every specifiable table column. You cannot use defined default values, but you can specify an empty string as a value. Because the RowID column is not specifiable, do not include a RowID value in the VALUES list.

This statement inserts a row of four values into the `Sample.Address` table in order. The statement assumes that the table contains exactly four columns whose data corresponds to the values being inserted, for example: `Street`, `City`, `State`, and `ZipCode`.

SQL

```
INSERT INTO Sample.Address VALUES ('22 Main St.','Anytown','PA','65342')
```

Example: Insert Rows into Table Using Specified Values

- **INSERT INTO** *table* **SET** *column = value*, *column2 = value2*, *...* inserts a row of values by explicitly setting the values of specific columns.

  This statement performs the same operation as in the **INSERT INTO** *table* (*column*, *column2*, *...*) **VALUES** (*value*, *value2*, *...*) syntax.

  SQL

  ```
  INSERT INTO Sample.Records SET StatusDate='05/12/22',Status='Purged'
  ```

  Example: Insert Rows into Table Using Specified Values

- **INSERT INTO** *table* **DEFAULT VALUES** inserts a row into a table that contains only default column values.

  – Columns with a defined default value are set to that value.

  – Columns without a defined default value are set to NULL.

  This statement inserts a row of default column values into the `Sample.Person` table.

  SQL

  ```
  INSERT INTO Sample.Person DEFAULT VALUES
  ```

  Columns defined with the NOT NULL constraint and no defined DEFAULT fail this operation with an SQLCODE -108.

  Columns defined with the UNIQUE constraint can be inserted using this statement. If a column is defined with a UNIQUE constraint and no DEFAULT value, repeated DEFAULT VALUES class insert multiple rows with this UNIQUE column set to NULL. If a column is defined with a UNIQUE constraint and a DEFAULT value, you can use this statement only once. A second call fails with an SQLCODE -119 error.

  DEFAULT VALUES inserts a row with a system-generated integer values for counter columns, including the RowID column, IDENTITY column, SERIAL (%Counter) column, and ROWVERSION column.

- **INSERT INTO** *table* **VALUES :***array*() inserts values from an array, specified as a host variable, into the columns of a table. You can use this syntax with Embedded SQL only. The values in this array must implicitly correspond to the columns of the row in column number order. You must specify a value for each specifiable column. An **INSERT** using column order cannot take defined column default values.

  This statement populates the array at runtime, enabling you to delay specifying which columns to insert until runtime. All other inserts require that you specify which columns are to be inserted when the **INSERT** is prepared. This syntax cannot be used with a linked table. Attempting to do so results in an SQLCODE -155 error.

  This class method uses embedded SQL to insert an array into the Sample.FullName table. `myarray(1)` is reserved for the RowID column and is therefore not specified.

Class Member

```
ClassMethod EmbeddedSQLInsertHostVarArray()
{
  set myarray(2)="Juanita"
  set myarray(3)="Pybus"
  &sql(INSERT INTO Sample.FullName VALUES :myarray())
  if SQLCODE '= 0 {
    write !, "Insert failed, SQLCODE= ", SQLCODE, ! ,%msg
    quit
  }
  write !,"Insert succeeded" quit
}
```

For more details on host variables and arrays, see Host Variable as a Subscripted Array.

Examples:

– Insert Data Using Embedded SQL

– Insert Stream Data into Table

Multi-Row Inserts

- **INSERT INTO** *table query* inserts rows of data that come from the result set of a SELECT query. The columns in the result set must match the columns in the table. You can use **INSERT** with a **SELECT** to populate a table with existing data extracted from other tables.

  This statement populates a table, `Sample.DupTable`, with the same values as the source table, `Sample.SrcTable`. The two tables must have the same number of columns, the same column names, and the same column order.

  SQL

  ```
  INSERT INTO Sample.DupTable SELECT * FROM Sample.SrcTable
  ```

  Example: Insert Data from Another Table Using a SELECT Query

- **INSERT INTO** *table* (*column*, *column2*, *...*) *query* inserts rows of data from the query result set into the specified columns. The **INSERT** statement sets unspecified column values in the row to NULL or to their default values.

  This statement inserts the query result set data from the `Name` and `DOB` columns of `Sample.Person` into the matching columns of the `Sample.Kids` table.

  SQL

  ```
  INSERT INTO Sample.Kids (Name,DOB) SELECT Name,DOB FROM Sample.Person WHERE Age <= 18
  ```

  Example: Insert Data from Another Table Using a SELECT Query

Insert Options

- **INSERT** *table* **...** omits the **INTO** keyword.

- **INSERT** *%keyword* **[INTO]** *table* **...** sets one or more *%keyword* options, separated by spaces. Valid options are %NOCHECK, %NOFPLAN, %NOINDEX, %NOJOURN, %NOLOCK, %NOTRIGGER, %PROFILE, and %PRO-FILE_ALL.

Arguments

table

The name of the table or view on which to perform the insert operation. A table or view name can be qualified (schema.table), or unqualified (table). An unqualified name is matched to its schema using either a schema search path, if specified, or the default schema name.

You can also perform **INSERT** using a subquery in place of the *table* argument. For example:

SQL

```
INSERT INTO (SELECT column1 AS c1 FROM MyTable) (c1) VALUES ('test')
```

To insert a row into a table, you must have appropriate table-level privileges.

column

A column name or comma-separated list of column names that correspond in sequence to the supplied list of values. If omitted, the list of values is applied to all columns in column-number order.

To insert a row into a table, you must have appropriate column-level privileges.

value

A scalar expression, or comma-separated list of scalar expressions, specified in the VALUES clause that supplies the data values for the corresponding columns in *column*. Specifying fewer values than columns generates an SQLCODE -62 error. Specifying more values than columns generates an SQLCODE -116 error.

To insert a row into a table, the column values specified in *value* must meet these requirements:

- Each column value must pass data type validation. Attempting to insert a column value inappropriate to the column data type results in an SQLCODE -104 error. This requirement applies only to an inserted data value. A column that takes a DEFAULT value does not have to pass data type validation or data size validation. The data type of the column, not the data type of the inserted data value, determines appropriateness. For example, attempting to insert a string value into a date column fails unless the string passes date validation for the current mode. However, attempting to insert a date value into a string column succeeds. **INSERT** inserts the date into the table as a literal string. To convert data to the destination data type, use the CONVERT function.

- Each data value must be within the MAXLEN, MAXVAL, and MINVAL for the column. For example, attempting to insert a string longer than 24 characters into a column defined as VARCHAR(24), or attempting to insert a number larger than 127 into a column defined as TINYINT, results in an SQLCODE -104 error.

- Supplying an invalid DOUBLE number via ODBC or JDBC results in an SQLCODE -104 error.

- Inserted data values must pass display to logical mode conversion. InterSystems SQL stores data in *logical* mode format. For some data types, the logical format might differ from the display format. For example, date data is stored as an integer count of days, time data is stored as a count of seconds from midnight, and %List data is stored as an encoded string. Other data types, such as strings and numbers, require no conversion. Attempting to insert a value in a format that cannot be converted to its logical storage value results in an error. For more details on mode conversions, see Data Display Options

- Every data value must pass data constraint validation for the column it is being inserted into.

  - A column defined as NOT NULL must be provided with a data value. If the column has no DEFAULT value, not specifying a data value results in an SQLCODE -108 error.

  - Data values must obey UNIQUE data constraints defined on a column or group of columns. Attempting to insert duplicate values into a unique column (such as the primary key column) or a unique column group results in an

SQLCODE -119 error. This error can also occur when you do not specify a value and a second use of the column's DEFAULT would supply a duplicate value.

– A column defined as a persistent class property with the VALUELIST parameter can accept only values listed in VALUELIST or no value (NULL). VALUELIST values are case-sensitive. Specifying a data value that does not match the VALUELIST values results in an SQLCODE -104 column error.

- Numbers are inserted in canonical form but can be specified with leading and trailing zeros and multiple leading signs. However, in SQL, two consecutive minus signs are parsed as a single-line comment indicator. Therefore, attempting to specify a number with two consecutive leading minus signs results in an SQLCODE -12 error.

- By default, an insert cannot specify values for columns for which the value is system-generated, such as the RowID, IDKey, or IDENTITY column. By default, attempting to insert a non-NULL value for any of these columns results in an SQLCODE -111 error. Attempting to insert a NULL for one of these columns causes InterSystems IRIS to override the NULL with a system-generated value, which does not produce an error.

  – If the table defines a ROWVERSION column, that column is automatically assigned a system-generated counter value when a row is inserted. Attempting to insert a value into a ROWVERSION column results in an SQLCODE -138 error.

  – An IDENTITY column can be made to accept user-specified values. By setting the **SetOption("IdentityInsert")** method, you can override the IDENTITY column default constraint and allow inserts of unique integer values to IDENTITY columns. To return the current setting for this constraint, call the **GetOption("IdentityInsert")** method. Inserting an IDENTITY column value changes the IDENTITY counter so that subsequent system-generated values increment from this user-specified value. Attempting to insert a NULL for an IDENTITY column generates an SQLCODE -108 error.

  – IDKey data has the following restriction. Because multiple IDKey columns in an index are delimited using the "||" (double vertical bar) characters, you cannot insert data values with this character string into IDKey columns.

- Values being inserted must not violate foreign key referential integrity, unless the **INSERT** command specifies the %NOCHECK keyword or the foreign key was defined with the NOCHECK keyword. Otherwise, attempting an insert that violates foreign key referential integrity results in an SQLCODE -121 error. For details on listing a table's foreign key constraints and the naming of foreign key constraints, refer to Catalog Details: Constraints.

- A data value cannot be a subquery. Attempting to specify a subquery as a column value results in an SQLCODE -144 error.

Non-Display Character Values

To insert values containing non-display characters, use the CHAR function and the concatenation operator. For example, this statement inserts a string consisting of the letter "A", a line feed character, and the letter "B":

SQL

```
INSERT INTO MyTable (Text) VALUES ('A'||CHAR(10)||'B')
```

To concatenate the results of a function, you must use the || concatenation operator, not the _ concatenation operator used in ObjectScript.

Special Variable Values

You can specify *value* as one of these special variables:

- A %TABLENAME or %CLASSNAME pseudo-column variable keyword. %TABLENAME returns the current table name. %CLASSNAME returns the name of the class corresponding to the current table.

- One or more of these ObjectScript special variables, including their abbreviated forms: $HOROLOG, $JOB, $NAMESPACE, $TLEVEL, $USERNAME, $ZHOROLOG, $ZJOB, $ZNSPACE, $ZPI, $ZTIMESTAMP, $ZTIMEZONE, $ZVERSION.

List Values

InterSystems IRIS supports the list structure data type, %List, data type class %Library.List. This compressed binary format does not map to a corresponding native data type for InterSystems SQL. Instead, it corresponds to data type VARBINARY with a default MAXLEN of 32749. For this reason, Dynamic SQL cannot use **INSERT** or **UPDATE** to set a property value of type %List. For more details, see Data Types.

IDENTITY and Counter Values

InterSystems SQL enables you to define columns that are system-generated, such as the IDENTITY column, or that automatically increment upon each **INSERT** or **UPDATE** operation, such as the RowVersion, AutoIncrement, and Serial Counter columns.

If you insert a value into one of these columns, the INSERT operation might fail, depending on the column type.

Column Type	INSERT Allowed?
IDENTITY	Not by default.
	To configure IDENTITY to accept inserted values, set the %CLASSPARAMETER ALLOWIDENTITYINSERT=1 value when defining the table. For more details, see Creating Named RowId Column Using IDENTITY Keyword.
ROWVERSION	No user-specified, calculated, or default value can be inserted for a ROWVERSION column.
SERIAL	Yes.
AUTO_INCREMENT	If you specify a positive integer value, **INSERT** inserts the value into the column, overriding the default counter value.
	If you specify no value, 0 (zero), or a nonnumeric value, **INSERT** ignores the specified value, increments this column's value by 1, and inserts that value into the column.

Computed Values

You can insert a value into a column with a defined COMPUTECODE under these conditions:

Defined Column	Value Behavior
COMPUTECODE with no related compute keywords	Value is computed and stored on **INSERT**. Value is not changed on **UPDATE**.
COMPUTECODE with COMPUTEONCHANGE	Value is computed and stored on **INSERT**. Value is recomputed and stored on **UPDATE**.
COMPUTECODE with DEFAULT and COMPUTEONCHANGE	Default value is stored on **INSERT**. Value is computed and stored on **UPDATE**.
COMPUTECODE with CALCULATED or TRANSIENT	If you insert a valid value into a calculated column, InterSystems IRIS inserts the row and increments ROWCOUNT. However, because this value is not stored, it is not inserted. When you query this column, InterSystems SQL recomputes the value and returns that value. If a column of this type is part of a foreign key constraint, a value for this column is computed during the insert in order to perform the referential integrity check. This computed value is not stored.

If the compute code contains a programming error (for example, divide by zero), the INSERT operation fails with an SQLCODE -415 error.

For more details, see Computing a field value on INSERT or UPDATE.

query

A SELECT query, the result set of which supplies the data values for the corresponding columns specified in *column*.

The **SELECT** query extracts column data from one or more tables and the **INSERT** command creates corresponding new rows in its table containing this column data. Corresponding columns can have different column names and column lengths, so long as the inserted data can fit in the table column. If the corresponding columns do not pass data type and length validation checks, InterSystems SQL generates an SQLCODE -104 error.

To limit the number of rows inserted, specify a TOP clause in the SELECT statement. To determine which of these top rows the query selects, use an ORDER BY clause in the **SELECT** statement.

To insert only unique values of certain columns, specify a GROUP BY clause in the query. By default, **GROUP BY** converts values to uppercase for the purpose of grouping. To preserve the letter case of inserted values, specify %EXACT collation in the query. For example:

SQL

```
INSERT INTO Sample.UniquePeople (Name,Age)
    SELECT Name,Age FROM Sample.Person
    WHERE Name IS NOT NULL GROUP BY %EXACT Name
```

An **INSERT** with **SELECT** operation sets the %ROWCOUNT variable to the number of rows inserted (either 0 or a positive integer).

array

A dynamic local array of values specified as a host variable. This value applies to Embedded SQL only.

The lowest subscript level of the array must be unspecified. Thus `:myupdates()`, `:myupdates(5,)`, and `:myupdates(1,1,)` are all valid specifications.

%keyword

Keyword options that configure **INSERT** processing. You can specify keyword options in any order. Separate multiple keyword options by spaces.

This table describes the keyword options that you can specify:

Keyword Option	**Description**

Keyword Option	Description
%NOCHECK	Disable unique value checking and foreign key referential integrity checking. %NOCHECK also disables validation for column data types, maximum column lengths, and column data constraints. When performing an **INSERT** through a view, the WITH CHECK OPTION validation is not performed. **Note:** %NOCHECK inserts can result in invalid data. If you enable this option, such as for speeding up bulk inserts or updates, make sure the data is from a reliable source. This option requires setting the corresponding %NOCHECK administrative privilege. Failing to set this privilege generates an SQLCODE -99 error on insert. To prevent inserts of non-unique data values when specifying %NOCHECK, perform an EXISTS check prior to **INSERT**. To disable foreign key referential integrity checking, use the **$SYSTEM.SQL.SetFilerRefIntegrity()** method instead. Alternatively, define a foreign key on the table by using the NOCHECK keyword so that foreign key referential integrity checking is never performed. For more details on foreign key referential integrity, see Foreign Key Referential Integrity Checking.
%NOFPLAN	Ignore any frozen plans for this operation and generate a new query plan. The frozen plan is retained but not used. For more details, see Frozen Plans.
%NOINDEX	Disable setting of index maps during **INSERT** processing. This option requires setting the corresponding %NOINDEX administrative privilege. Failing to set this privilege generates an SQLCODE -99 error on insert. To build the index of a table containing rows that were not indexed upon insertion, use BUILD INDEX.
%NOJOURN	Suppress journaling and disables transactions for the duration of the insert operation. None of the changes made in any of the rows are journaled, including any triggers pulled. If you perform a **ROLLBACK** after a statement with %NOJOURN, the changes made by the statement are not rolled back. This option requires setting the corresponding %NOJOURN administrative privilege. Failing to set this privilege generates an SQLCODE -99 error on insert.
%NOLOCK	Disable locking of the row upon **INSERT**. Set this option only when a single user or process is updating the database. This option requires setting the corresponding %NOLOCK administrative privilege. Failing to set this privilege generates an SQLCODE -99 error on insert.
%NOTRIGGER	Do not pull base table insert triggers during **INSERT** processing. This option requires setting the corresponding %NOTRIGGER administrative privilege. Failing to set this privilege generates an SQLCODE -99 error on insert.

Keyword Option	Description
%PROFILE	Generate performance analysis statistics (SQLStats) for the insert statement.
%PROFILE_ALL	• %PROFILE collects SQLStats for the main query module • %PROFILE_ALL collects SQLStats for the main query module and all of its subquery modules The generated statements are the same generated with the SQL Performance Analysis Toolkit enabled. This keyword option enables you to profile and inspect individual statements, leaving statistics disabled for other compiled statements that do not require investigation. For more details on these statistics, see SQL Runtime Statistics.

Examples

Insert Rows into Table Using Specified Values

This example shows that various ways that you can insert a new row of values into a table.

Create a table containing company data. This table has two required columns: the name of the company, which must be unique, and the country in which the company headquarters is located. The second column, Revenue, is not required and has a default value of 0.

SQL

```
CREATE TABLE Sample.Company (
    Name VARCHAR(20) UNIQUE NOT NULL,
    Revenue INTEGER DEFAULT 0,
    Country VARCHAR(10) NOT NULL)
```

Insert a row of data into the table. The column values must be specified in the same order as the table's column order.

SQL

```
INSERT INTO Sample.Company VALUES ('CompanyA',10000,'BEL')
```

Insert another row of data, this time specifying the column names to insert data into. Because this statement omits the Revenue column, InterSystems SQL sets this column value to its default value of 0.

SQL

```
INSERT INTO Sample.Company (Name,Country) VALUES ('CompanyB','CAN')
```

Insert a third row of data, this time using the SET *column=value* syntax. Note that the *column=value* pairs do not have to be in table column order.

SQL

```
INSERT INTO Sample.Company Set Name = 'CompanyC', Country = 'ECU', Revenue = 25000
```

View the inserted data, ordered by their revenue.

SQL

```
SELECT * FROM Sample.Company ORDER BY Revenue DESC
```

Name	Revenue	Country
CompanyC	25000	ECU
CompanyA	10000	BEL
CompanyB	0	CAN

Delete the table when you are done.

SQL

```
DROP TABLE Sample.Company
```

Insert Stream Data into Table

These examples show the different types of data values you can insert into a stream field by using embedded SQL.

For any table, you can insert a string literal or a host variable containing a string literal. For example:

ObjectScript

```
set literal="Technique 1"
&sql(INSERT INTO MyStreamTable (MyStreamField) VALUES (:literal))
```

You can also insert an object reference (OREF) into a stream object for any non-sharded table. InterSystems IRIS opens this object and copies its contents into the new stream field. For example:

ObjectScript

```
set oref=##class(%Stream.GlobalCharacter).%New()
do oref.Write("Technique non-shard 1")

//do the insert; use an actual OREF
&sql(INSERT INTO MyStreamTable (MyStreamField) VALUES (:oref))
```

Alternatively, you can insert a string version of an OREF into a stream object:

ObjectScript

```
set oref=##class(%Stream.GlobalCharacter).%New()
do oref.Write("Technique non-shard 2")

//next line converts OREF to a string OREF
set string=oref_""

//do the insert
&sql(INSERT INTO MyStreamTable (MyStreamField) VALUES (:string))
```

For a sharded table, you can insert an object ID (OID) using a temporary stream object stored in the ^IRIS.Stream.Shard global:

ObjectScript

```
set clob=##class(%Stream.GlobalCharacter).%New("Shard")
do clob.Write("Technique Sharded Table 1")
set sc=clob.%Save() // Handle $$$ISERR(sc)
set ClobOid=clob.%Oid()

&sql(INSERT INTO MyStreamTable (MyStreamField) VALUES (:ClobOid))
```

Attempting to insert an improperly defined stream value results in an SQLCODE -412 error.

For more details and examples, see Inserting Data into Stream Data Fields.

Insert Data Using Embedded SQL

This Embedded SQL example uses a host variable array to insert a row with three column values. Array elements are numbered in column order. Specified array values must start with the second element, in this case `company(2)`. The first array element corresponds to the RowID column, which is automatically supplied and cannot be defined:

ObjectScript

```
SET company(2)="Company1"
SET company(3)=15000
SET company(4)="JPN"
&sql(INSERT INTO Sample.Company VALUES :company())
```

This embedded SQL example uses a dynamic local array with an unspecified last subscript to pass an array of values to **INSERT** at runtime.

ObjectScript

```
NEW SQLCODE,%ROWCOUNT,%ROWID
&sql(INSERT INTO Sample.Employee VALUES :emp('profile',))
WRITE !,"SQL Error code: ",SQLCODE," Row Count: ",%ROWCOUNT
```

The previous statements causes each column in the inserted "Employee" row to be set to the following, where "col" is the column's number in the Sample.Employee table.

```
emp("profile",col)
```

Insert Data Using Dynamic SQL

This class method uses Dynamic SQL to insert values into a table based on arguments passed into the method.

Class Member

```
ClassMethod DynamicSQLInsert(name As %String, revenue As %Integer, country As %String)
{
  set sqltext = "INSERT INTO Sample.Company (Name,Revenue,Country) VALUES (?,?,?)"

  set tStatement = ##class(%SQL.Statement).%New(0,"Sample")
  set qStatus = tStatement.%Prepare(sqltext)
    if qStatus'=1 {write "%Prepare failed:" DO $System.Status.DisplayError(qStatus) quit}
  set rtn = tStatement.%Execute(name,revenue,country)
  if rtn.%SQLCODE=0 {
    write !,"Insert succeeded"
    write !,"Row count=",rtn.%ROWCOUNT
    write !,"Row ID=",rtn.%ROWID }
  elseif rtn.%SQLCODE=-119 {
    write !,"Duplicate record not written",!,rtn.%Message  quit }
  else {  write !,"Insert failed, SQLCODE=",rtn.%SQLCODE  }
}
```

Insert Data from Another Table Using a SELECT Query

This example shows how to populate a table with data extracted from another table by using an **INSERT** with **SELECT** operation. This example assumes that you have a previously defined Sample.Person table that contains Name, DOB, and Age columns. You can download such a table from GitHub at https://github.com/intersystems/Samples-Data. For download instructions, see Downloading Samples for Use with InterSystems Products.

Create a table called MyStudents. This table contains name and date-of-birth columns, both of which are specified, and an age column, which is calculated from the date-of-birth column.

SQL

```
CREATE TABLE MyStudents (
   StudentName VARCHAR(32),
   StudentDOB DATE,
   StudentAge INTEGER COMPUTECODE {set {StudentAge} =
                       $piece(($piece($horolog,",",1)-{StudentDOB})/365,".",1)}
                       CALCULATED)
```

Insert student data from the Sample.Person table into the MyStudents table. Use a SELECT query that selects the people that are 21 and under. You can use either of these two queries. Because the column order of the two tables match and only the first two columns are stored, the column names can be omitted.

SQL

```
INSERT INTO MyStudents (StudentName,StudentDOB)
   SELECT Name,DOB
   FROM Sample.Person WHERE Age <= 21
```

SQL

```
INSERT INTO MyStudents
   SELECT Name,DOB
   FROM Sample.Person WHERE Age <= 21
```

Display the results, ordered by age.

SQL

```
SELECT * FROM MyStudents ORDER BY StudentAge
```

Delete the table when you are done.

SQL

```
DROP TABLE MyStudents
```

Another use of **INSERT** with **SELECT** is to create a duplicate table from an existing table. You can use this operation to copy existing data into a redefined table that will accept future column data values that would not have been valid in the original table. For more details, see Copy Data into a Duplicate Table.

Compatibility

To use **INSERT** to add data to an InterSystems IRIS table using Microsoft Access, either mark the table RowID column as private or define a unique index on one or more additional columns.

Security and Privileges

Table-Level Privileges

To insert one or more rows of data into a table, you (or the specified user) must have either table-level privileges or column-level privileges for that table.

- When inserting any data into a table, you must have INSERT privilege on the table.

- If you are inserting data from another table using a SELECT query, you must have SELECT privilege on that table.

The Owner (creator) of the table is automatically granted all privileges for that table. If you are not the Owner, you must be granted privileges for the table. Failing to do so results in an SQLCODE -99 error.

To determine if you have the appropriate privileges, use %CHECKPRIV. To assign table privileges to a user, use GRANT. For more details, see Privileges.

To insert into a sharded table, you must have INSERT privileges for the target tables. Failing to have these privileges results in an SQLCODE -253 error.

Table-level privileges are equivalent to, but not identical to, having column-level privileges on all columns of the table.

Column-Level Privileges

If you do not have table-level INSERT privilege, to insert a specified value into a column, you must have column-level INSERT privilege for that column. Only those columns for which you have INSERT privilege receive the value specified in the **INSERT** command.

If you do not have column-level INSERT privilege for a specified column, InterSystems SQL inserts the column's default value (if defined), or NULL (if no default is defined). If you do not have INSERT privilege for a column that has no default and is defined as NOT NULL, InterSystems IRIS issues an SQLCODE -99 (Privilege Violation) error at Prepare time.

If the **INSERT** command specifies columns in the WHERE clause of a **SELECT** subquery, you must have these privileges:

- SELECT privilege for those columns if they are not data insert columns.

- Both SELECT and INSERT privileges for those columns if they are included in the result set.

When a property is defined as ReadOnly, the corresponding table column is also defined as ReadOnly. You can assign a value to a ReadOnly column only by using InitialExpression or SqlComputed. Attempting to insert a value into column for which you have column-level ReadOnly (SELECT or REFERENCES) privilege results in an SQLCODE -138 error.

To determine if you have the appropriate privileges, use %CHECKPRIV. To assign column-level privileges to a user, use GRANT. For more details, see Privileges.

Row-Level Security

InterSystems IRIS row-level security permits **INSERT** to add a row even if the row security is defined so that you will not be permitted to subsequently access the row. To ensure that an **INSERT** does not prevent you from subsequent **SELECT** access to the row, perform the **INSERT** through a view that has a WITH CHECK OPTION. For more details, see CREATE VIEW.

Fast Insert

When inserting rows in a table using JDBC or ODBC, InterSystems IRIS by default automatically performs highly efficient Fast Insert operations. Fast Insert moves the normalization and formatting of the data being insert from the server over to the client. The server can then directly set the whole row of data for a table into the global without manipulation on the server. This offloads these tasks from the server onto the client and can dramatically improve INSERT performance. Because the client is assuming the task of formatting the data, there may be an unforeseen usage increase in your client environment. You can use the FeatureOption property to disable Fast Insert if this is an issue.

Fast Insert must be supported on both the server and the client. To enable or disable Fast Insert in the client, use the FeatureOption property in the definition of the class instance as follows:

```
Properties p = new Properties();
p.setProperty("FeatureOption","3"); // 2 is fast Insert, 1 is fast Select, 3 is both
```

If Fast Insert is active, an **INSERT** executed using a cached query is performed using Fast Insert. This initial **INSERT** that generated the cached query is not performed using Fast Insert. This enables you to compare the performance of the initial insert with subsequent Fast Inserts executed using the cached query. If Fast Insert is not supported (for any of the following reasons), an ordinary **INSERT** is performed.

Fast Insert must be performed on a table. It cannot be performed on an updateable view. Fast Insert is not performed when the table has any of the following characteristics:

- The table uses embedded (nested) storage structure (%SerialObject).

- The table is a linked table.

- The table is a child table.

- The table has an explicitly defined multi-field IDKEY index.

- The table has a SERIAL (%Counter), AUTO INCREMENT, or %RowVersion field.

- The table has a property (field) with a defined VALUELIST parameter.

- The table has a defined insert trigger.

- The table performs LogicalToStorage conversion of field values.

- The table is a Shard Master table.

Fast Insert cannot be performed if the INSERT statement has any of the following characteristics:

- It specifies a stream field (data type %Stream.GlobalCharacter or %Stream.GlobalBinary), a collection field (lists or arrays), or a ReadOnly field. These types of fields can exist in the table, but cannot be specified in the **INSERT**.

- It specifies a literal value enclosed with double parentheses that suppresses literal substitution. For example, (('A')).

- It specifies a {ts } timestamp value that omits the date value.

- It includes a DEFAULT VALUES clause.

For SQL xDBC Statement auditing events, an **INSERT** statement that uses the Fast Insert interface has a description of `SQL fastINSERT Statement`. If the Fast Insert interface is used, the Audit event does not include any parameter data, but includes the message `Parameter values are not available for a fastInsert statement`.

ODBC Datatype Handling

When using Fast Insert using ODBC, you may set fields TIMESTAMP or POSIX typed fields using the integer or _int64 data types. The provided number will be treated as a $HOROLOG value.

String values that are converted to doubles will be validated, ensuring that the String represents a numeric value. This validation also accepts, in any case, "INF", "infinity", and "NaN" values, which may be preceded by a + or - sign.

Values being converted from Numeric to integer will be truncated, not rounded.

Transaction Considerations

Transaction Atomicity Settings

By default, **INSERT**, **UPDATE**, **DELETE**, and **TRUNCATE TABLE** are atomic operations. An **INSERT** either completes successfully or the whole operation is rolled back. If any of the specified rows cannot be inserted, none of the specified rows are inserted and the database reverts to its state before issuing the **INSERT**.

You can modify this default for the current process within SQL by invoking SET TRANSACTION %COMMITMODE. You can modify this default for the current process in ObjectScript by invoking the **SetOption()** method, using this syntax:

```
SET status=$SYSTEM.SQL.Util.SetOption("AutoCommit",intval,.oldval)
```

The following *intval* integer options are available:

- 1 or IMPLICIT (autocommit on — default) — Each **INSERT** constitutes a separate transaction.

- 2 or EXPLICIT (autocommit off) — If no transaction is in progress, **INSERT** automatically initiates a transaction, but you must explicitly **COMMIT** or **ROLLBACK** to end the transaction. In EXPLICIT mode, the number of database operations per transaction is user-defined.

- 0 or NONE (no auto transaction) — No transaction is initiated when you invoke **INSERT**. A failed **INSERT** operation can leave the database in an inconsistent state, with some rows inserted and some not inserted. To provide transaction

support in this mode, you must use **START TRANSACTION** to initiate the transaction and **COMMIT** or **ROLLBACK** to end the transaction.

A sharded table is always in no auto-transaction mode, which means all inserts, updates, and deletes to sharded tables are performed outside the scope of a transaction.

To determine the atomicity setting for the current process, use the **GetOption("AutoCommit")** method, as shown in this ObjectScript example:

ObjectScript

```
SET stat=$SYSTEM.SQL.Util.SetOption("AutoCommit",$RANDOM(3),.oldval)
IF stat'=1 {WRITE "SetOption failed:" DO $System.Status.DisplayError(qStatus) QUIT}
   SET x=$SYSTEM.SQL.Util.GetOption("AutoCommit")
   IF x=1 {
     WRITE "Default atomicity behavior",!
     WRITE "automatic commit or rollback" }
   ELSEIF x=0 {
     WRITE "No transaction initiated, no atomicity:",!
     WRITE "failed DELETE can leave database inconsistent",!
     WRITE "rollback is not supported" }
   ELSE { WRITE "Explicit commit or rollback required" }
```

Modify Transaction Lock Threshold

If the %NOLOCK keyword is not specified, the system automatically performs standard record locking on **INSERT**, **UPDATE**, and **DELETE** operations. Each affected record (row) is locked for the duration of the current transaction.

The default lock threshold is 1000 locks per table. If you insert more than 1000 records from a table during a transaction, the lock threshold is reached and InterSystems IRIS automatically escalates the locking level from record locks to a table lock. This permits large-scale inserts during a transaction without overflowing the lock table.

InterSystems IRIS applies one of these lock escalation strategies:

- "E"-type escalation locks — InterSystems IRIS uses this lock escalation if the following are true:

  1. The class of the table uses %Storage.Persistent. You can determine this from the Catalog Details in the Management Portal SQL schema display.

  2. The class either does not specify an IDKey index, or specifies a single-property IDKey index.

  For details on "E"-type lock escalation, see LOCK.

- Traditional SQL lock escalation — This lock escalation can take place when the class has multi-property IDKey index. In this case, each %Save increments the lock counter. This means if you do 1001 saves of a single object within a transaction, InterSystems IRIS attempts to escalate the lock.

For both lock escalation strategies, you can determine the current system-wide lock threshold value using the **$SYSTEM.SQL.Util.GetOption("LockThreshold")** method. The default is 1000. You can configure the system-wide lock threshold using one of these options:

- Call the **$SYSTEM.SQL.Util.SetOption("LockThreshold")** method.

- In the Management Portal, select **System Administration**, **Configuration**, **SQL and Object Settings**, **SQL**. Display and edit the current setting of **Lock escalation threshold**. The default is 1000 locks. If you change this setting, any new process started after changing it will have the new setting.

To change the lock threshold, you must have USE permission on the %Admin Manage Resource. InterSystems IRIS immediately applies any change made to the lock threshold value to all current processes.

One potential consequence of automatic lock escalation is a deadlock situation that might occur when an attempt to escalate to a table lock conflicts with another process holding a record lock in that table. There are several possible strategies to avoid this:

---

1. Increase the lock escalation threshold so that lock escalation is unlikely to occur within a transaction.

2. Substantially lower the lock escalation threshold so that lock escalation occurs almost immediately, thus decreasing the opportunity for other processes to lock a record in the same table.

3. Apply a table lock for the duration of the transaction and do not perform record locks. This can be done at the start of the transaction by specifying LOCK TABLE, then UNLOCK TABLE (without the IMMEDIATE keyword, so that the table lock persists until the end of the transaction), then perform inserts with the %NOLOCK option.

Automatic lock escalation is intended to prevent overflow of the lock table. However, if you perform such a large number of inserts that a <LOCKTABLEFULL> error occurs, **INSERT** issues an SQLCODE -110 error.

For further details on transaction locking, see Transaction Processing.

Child Table Insert

During an **INSERT** operation on a child table, a shared lock is acquired on the corresponding row in the parent table. This row is locked while inserting the child table row. The lock is then released (it is not held until the end of the transaction). This ensures that the referenced parent row is not changed during the insert operation.

More About

Copy Data into a Duplicate Table

You can use **INSERT** with **SELECT *** to copy the data from one table into a duplicate table, as long as the column order matches and the data types are compatible. Use this operation to copy existing data into a redefined table that will accept future column data values that would not have been valid in the original table. Sample syntax:

SQL

```
INSERT INTO Sample.DupTable SELECT * FROM Sample.SrcTable
```

The column names do not have to match, but the data in the source and destination tables must meet these requirements:

- The data type of the source table values must be compatible with the data type of the destination table columns. You can, for example, insert integer data from an INTEGER column into a VARCHAR column, because the INTEGER value can be converted to a VARCHAR. If any data value is incompatible, the **INSERT** fails with an SQLCODE -104 error. To explicitly convert inserted data to the destination data type, you can use the CONVERT function.

- The data type length of the source table values must be compatible with the length of the destination table columns. The defined column data lengths do not have to match each other, they just have to match the actual data. For example, suppose SrcTable has a `FullName VARCHAR(60)` column and DupTable has a corresponding `PersonName VARCHAR(40)` column. If no existing FullName value is longer than 40 characters, the **INSERT** succeeds. If any FullName is longer than 40 characters, the **INSERT** fails with an SQLCODE -104 error.

- The two tables must have a compatible column order or the INSERT command fails with an SQLCODE -64 error. The DDL **CREATE TABLE** operation lists the columns in the order defined. A persistent class that defines a table lists the columns in alphabetical order.

- The tables must have a compatible column count, but the destination table can have additional columns beyond the ones copied. For example, SrcTable can have the columns `FullName VARCHAR(60)` and `Age INTEGER`, and DupTable can have the columns `PersonName VARCHAR(60)`, `Years INTEGER`, and `ShoeSize INTEGER`.

- If either the source and destination table defines a public RowID, copying data is restricted, as shown in this table:

Source Table	Destination Table	Copy Operation Behavior
Private RowID	Private RowID	You can use INSERT SELECT with SELECT * to copy data to a duplicate table.
Public RowID	Public RowID	You cannot use INSERT SELECT to copy data to a duplicate table. An SQLCODE -111 error is generated.
Private RowID	Public RowID	You cannot use INSERT SELECT to copy data to a duplicate table. An SQLCODE -111 error is generated.
Public RowID	Private RowID	You cannot use INSERT SELECT with SELECT * to copy data to a duplicate table. An SQLCODE -64 error is generated because of the presence of the RowID in one select list makes the select lists incompatible. You can use an INSERT SELECT with a list of all column names (not including the RowID) to copy data to a duplicate table. However, if the Source has a foreign key public RowID, the foreign key relationship is not preserved for the destination table. The Destination receives new system-generated RowIDs. If the source table has a foreign key public RowID, and you want the destination table to have the same foreign key relationship, you must define the destination table with the `%CLASSPARAMETER ALLOWIDENTITYINSERT=1` in **CREATE TABLE**. If a table is defined as ALLOWIDENTITYINSERT=1, this setting cannot be changed by the **SetOption("IdentityInsert")** method.

The DDL **CREATE TABLE** operation defines the RowID as private by default. A persistent class that defines a table defines the RowID as public by default. To make it private, you must specify the SqlRowIdPrivate class keyword when defining the persistent class. However, a foreign key can only refer to a table with a public RowID.

If the persistent class of a source or destination table defines the Final keyword, this keyword has no effect on copying data into a duplicate table.

Insert Data into %SerialObject Properties

When inserting data into a %SerialObject, you must insert into the table (persistent class) that references the embedded %SerialObject. You cannot insert into a %SerialObject directly. For example, consider a persistent class has a property, PAddress, that references a serial object contain the properties Street, City, and Country, in that order. You can insert values of this property in these ways:

- Use the referencing field to insert values for multiple %SerialObject properties as a %List structure. For example:

SQL

```
INSERT INTO MyTable SET PAddress=$LISTBUILD('123 Main St.','Newtown','USA')
```

SQL

```
INSERT INTO MyTable (PAddress) VALUES ($LISTBUILD('123 Main St.','Newtown','USA'))
```

The %List must contain values for the properties of the serial object (or placeholder commas) in the order that these properties are specified in the serial object.

This type of insert might not perform validation of %SerialObject property values. Therefore, after inserting %Serialobject property values using a %List structure, use the **$SYSTEM.SQL.Schema.ValidateTable()** method to perform Table Data Validation.

- Use underscore syntax to insert values for individual %SerialObject properties in any order. For example:

SQL

```
INSERT INTO MyTable SET PAddress_City='Newtown',PAddress_Street='123 Main St.',PAddress_Country='USA'
```

Unspecified serial object properties default to NULL.

This type of insert performs validation of %SerialObject property values.

See Also

- INSERT OR UPDATE
- UPDATE
- DELETE
- CREATE TABLE
- JOIN
- SELECT
- VALUES
- Modifying the Database
- Defining Tables
- Defining Views
- Transaction Processing
- SQL and Object Settings Pages
- SQLCODE error messages

INSERT OR UPDATE (SQL)

Adds new rows or updates existing rows in a table.

Synopsis

Single Row Inserts or Updates

```
INSERT OR UPDATE table (column, column2, ...) VALUES (value, value2, ...)
INSERT OR UPDATE table VALUES (value, value2, ...)
INSERT OR UPDATE table SET column = value, column2 = value2, ...
INSERT OR UPDATE table DEFAULT VALUES
INSERT OR UPDATE table VALUES :array()
```

Multi-Row Inserts or Updates

```
INSERT OR UPDATE table query INSERT OR UPDATE table (column, column2, ...) query
```

Insert or Update Options

```
INSERT OR UPDATE INTO table ...
INSERT OR UPDATE %keyword [INTO] table ...
```

Description

The **INSERT OR UPDATE** command is an extension of the **INSERT** command, with these differences:

- If the row being inserted does not exist, **INSERT OR UPDATE** performs an **INSERT** operation.

- If the row being inserted already exists, **INSERT OR UPDATE** performs an **UPDATE** operation, updating the row with the specified column values. An update occurs even when the specified data values are identical to the existing data.

   An *existing row* is one in which the value being inserted already exists in a column that contains a unique constraint. For more details, see Uniqueness Checks.

INSERT OR UPDATE uses the same syntax and generally has the same features and restrictions as the **INSERT** statement. Special considerations for **INSERT OR UPDATE** are described on this page. Unless otherwise stated, see INSERT for more details.

Single Row Inserts or Updates

- **INSERT OR UPDATE** *table* (*column*, *column2*, ...) **VALUES** (*value*, *value2*, ...) inserts or updates a row of values for the specified columns of a table. The values in the VALUES clause must correspond positionally with the column names in the column list. The insert or update of a single row sets the %ROWCOUNT variable to 1 and the %ROWID variable to the inserted or updated row.

   This statement first tries inserting a new row of data into the Sample.Records table. If the RecordID column enforces a UNIQUE constraint and the RecordID being inserted already exists, then **INSERT OR UPDATE** updates the existing row instead.

   SQL

   ```
   INSERT OR UPDATE Sample.Records (RecordID,StatusDate,Status) VALUES (105,'05/12/22','Purged')
   ```

   Example: Insert or Update Rows in a Table

- **INSERT OR UPDATE** *table* **VALUES** (*value*, *value2*, ...) inserts or updates the table row of values in column number order. The data values must correspond positionally to the defined column list. You must specify a value for

every specifiable table column. You cannot use defined default values, but you can specify an empty string as a value. Because the RowID column is not specifiable, do not include a RowID value in the VALUES list.

This statement first tries inserting a row of four values into the `Sample.Address` table in order. If this combination of columns has a unique constraint, and a value for this key is already defined in the table, then **INSERT OR UPDATE** updates the existing row instead.

SQL

```
INSERT OR UPDATE Sample.Address VALUES ('22 Main St.','Anytown','PA','65342')
```

* **INSERT OR UPDATE** *table* **SET** *column = value*, *column2 = value2*, *...* inserts or updates a row of values by explicitly setting the values of specific columns.

This statement performs the same operation as in the **INSERT OR UPDATE** *table* **(***column***,** *column2***, ...) VALUES (***value***,** *value2***, ...)** syntax.

SQL

```
INSERT OR UPDATE Sample.Records SET RecordID=105, StatusDate='05/12/22',Status='Purged'
```

* **INSERT OR UPDATE** *table* **DEFAULT VALUES** inserts or updates a row that contains only default column values.

    – Columns with a defined default value are set to that value.

    – Columns without a defined default value are set to NULL.

This statement inserts a row of default column values into the `Sample.Person` table.

SQL

```
INSERT OR UPDATE Sample.Person DEFAULT VALUES
```

* **INSERT OR UPDATE** *table* **VALUES :***array***()** inserts or updates values from an array, specified as a host variable, into the columns of a table. You can use this syntax with Embedded SQL only. The values in this array must implicitly correspond to the columns of the row in column number order. You must specify a value for each specifiable column. An **INSERT OR UPDATE** using column order cannot take defined column default values.

This class method uses embedded SQL to insert into or update an array for the Sample.FullName table. `myarray(1)` is reserved for the RowID column and is therefore not specified.

Class Member

```
ClassMethod EmbeddedSQLInsertOrUpdateHostVarArray()
{
  set myarray(2)="Juanita"
  set myarray(3)="Pybus"
  &sql(INSERT OR UPDATE Sample.FullName VALUES :myarray())
  if SQLCODE '= 0 {
    write !, "Insert or update failed, SQLCODE= ", SQLCODE, ! ,%msg
    quit
  }
  write !,"Insert or update succeeded" quit
}
```

For more details on host variables and arrays, see Host Variable as a Subscripted Array.

Multi-Row Inserts or Updates

* **INSERT OR UPDATE** *table* *query* inserts or updates rows of data the come from the result set of a SELECT query. The columns in the result set must match the columns in the table. You can use **INSERT OR UPDATE** with a **SELECT** to populate a table with existing data extracted from other tables.

This statement inserts the `Name` row from the `Sample.Customer` table into the `Sample.Person` table, or updates existing rows of `Sample.Person` with the corresponding `Sample.Customer` values.

SQL

```
INSERT OR UPDATE Sample.Person SELECT Name FROM Sample.Customer
```

- **INSERT OR UPDATE** *table* (*column*, *column2*, *...*) *query* inserts or updates rows of data from the query result set into the specified columns.

  This statement inserts or updates the query result set data from the `Name` and `DOB` columns of `Sample.Person` into the matching columns of the `Sample.Kids` table.

  SQL

```
INSERT OR UPDATE Sample.Kids (Name,DOB) SELECT Name,DOB FROM Sample.Person WHERE Age <= 18
```

Insert or Update Options

- **INSERT OR UPDATE INTO** *table* **...** specifies the optional INTO keyword.
- **INSERT OR UPDATE** *%keyword* **[INTO]** *table* **...** sets one or more *%keyword* options, separated by spaces. Valid options are %NOCHECK, %NOFPLAN, %NOINDEX, %NOJOURN, %NOLOCK, %NOTRIGGER, %PROFILE, and %PROFILE_ALL.

  Note: Because the %NOCHECK keyword disables unique value checking, **INSERT OR UPDATE %NOCHECK** always results in an insert operation and is therefore equivalent to **INSERT**.

Arguments

table

The name of the table or view on which to perform the insert operation. This argument can also be a subquery.

column

A column name or comma-separated list of column names that correspond in sequence to the supplied list of values. If omitted, the list of values is applied to all columns in column-number order.

IDKEY column values can be inserted but not updated. For more details on this restriction, see IDKEY Column Values.

value

A scalar expression, or comma-separated list of scalar expressions, specified in the VALUES clause that supplies the data values for the corresponding columns in *column*. Specifying fewer values than columns generates an SQLCODE -62 error. Specifying more values than columns generates an SQLCODE -116 error.

INSERT OR UPDATE has the same value restrictions as **INSERT**. For more details, see the *value* argument of the **INSERT** command.

array

A dynamic local array of values specified as a host variable. This value applies to Embedded SQL only.

The lowest subscript level of the array must be unspecified. Thus `:myupdates()`, `:myupdates(5,)`, and `:myupdates(1,1,)` are all valid specifications.

query

A [SELECT](#) query, the result set of which supplies the data values for the corresponding columns specified in *column*.

The **SELECT** query extracts column data from one or more tables and the **INSERT OR UPDATE** command creates corresponding new rows in its table containing this column data. Corresponding columns can have different column names and column lengths, so long as the inserted data can fit in the table column. If the corresponding columns do not pass data type and length validation checks, InterSystems SQL generates an SQLCODE -104 error.

An **INSERT OR UPDATE** with **SELECT** operation sets the [%ROWCOUNT](#) variable to the number of rows inserted or updated (either 0 or a positive integer).

%keyword

Keyword options that configure **INSERT OR UPDATE** processing. You can specify keyword options in any order. Separate multiple keyword options by spaces.

You can specify these keywords:

- %NOCHECK — Disable unique value checking and foreign key referential integrity checking. Disables the update operation of **INSERT OR UPDATE**.

- %NOFPLAN — Ignore any frozen plans for this operation and generate a new query plan.

- %NOINDEX — Disable setting of index maps during **INSERT OR UPDATE** processing.

- %NOJOURN — Suppress journaling and turns off transactions for the duration of the insert operation.

- %NOLOCK — Disable locking of the row upon **INSERT OR UPDATE**.

- %NOTRIGGER — Do not pull base table [insert triggers](#) during **INSERT OR UPDATE** processing.

- %PROFILE, %PROFILE_ALL — Generate performance analysis statistics (SQLStats) for the **INSERT OR UPDATE** statement.

    - %PROFILE collects SQLStats for the main query module

    - %PROFILE_ALL collects SQLStats for the main query module and all of its subquery modules

For more details on these keywords, see the *[keyword](#)* argument of the **INSERT** command.

Examples

Insert or Update Rows in a Table

In this example, you create a new table (SQLUser.CaveDwellers), use **INSERT** to populate the table with data, and then use **INSERT OR UPDATE** to add additional rows and update existing rows.

Create a table with a column, Num, that is designated as the primary key. This constraint enforces column values to be unique and not null.

SQL

```
CREATE TABLE SQLUser.CaveDwellers (
Num INTEGER PRIMARY KEY,
CaveCluster CHAR(80) NOT NULL,
Troglodyte CHAR(50) NOT NULL)
```

Insert three rows into the table using an **INSERT OR UPDATE** statement, then use SELECT * to display the table data. Because the rows did not previously exist, **INSERT OR UPDATE** performs an insert operation for all of them.

SQL

```
INSERT OR UPDATE SQLUser.CaveDwellers (Num,CaveCluster,Troglodyte) VALUES (1,'Bedrock','Flintstone,Fred')
```

SQL

```
INSERT OR UPDATE SQLUser.CaveDwellers (Num,CaveCluster,Troglodyte) VALUES (2,'Bedrock','Flintstone,Wilma')
```

SQL

```
INSERT OR UPDATE SQLUser.CaveDwellers (Num,CaveCluster,Troglodyte) VALUES
(3,'Bedrock','Flintstone,Pebbles')
```

SQL

```
SELECT * FROM SQLUser.CaveDwellers
```

Num	CaveCluster	Troglodyte
1	Bedrock	Flintstone,Fred
2	Bedrock	Flintstone,Wilma
3	Bedrock	Flintstone,Pebbles

Insert or update four additional rows of data.

- For the first three statements, **INSERT OR UPDATE** performs an insert operation, because the values being inserted into the primary key column, Num, are not already in the table.

- For the last statements **INSERT OR UPDATE** performs an update operation, because the Num column value of 3 is already in the table. **INSERT OR UPDATE** updates the Troglodyte column with the new value for that row.

SQL

```
INSERT OR UPDATE SQLUser.CaveDwellers (Num,CaveCluster,Troglodyte) VALUES (4,'Bedrock','Rubble,Barney')
```

SQL

```
INSERT OR UPDATE SQLUser.CaveDwellers (Num,CaveCluster,Troglodyte) VALUES (5,'Bedrock','Rubble,Betty')
```

SQL

```
INSERT OR UPDATE SQLUser.CaveDwellers (Num,CaveCluster,Troglodyte) VALUES (6,'Bedrock','Rubble,Bamm-Bamm')
```

SQL

```
INSERT OR UPDATE SQLUser.CaveDwellers (Num,CaveCluster,Troglodyte) VALUES
(3,'Bedrock','Flintstone-Rubble,Pebbles')
```

SQL

```
SELECT * FROM SQLUser.CaveDwellers
```

Num	CaveCluster	Troglodyte
1	Bedrock	Flintstone,Fred
2	Bedrock	Flintstone,Wilma
3	Bedrock	Flintstone-Rubble,Pebbles
4	Bedrock	Rubble,Barney
5	Bedrock	Rubble,Betty
6	Bedrock	Rubble,Bamm-Bamm

Delete the table when you are done.

SQL

```
DROP TABLE SQLUser.CaveDwellers
```

Security and Privileges

INSERT OR UPDATE requires both INSERT and UPDATE privileges. You must have these privileges either as table-level privileges or as column-level privileges. For table-level privileges:

- You must have both INSERT and UPDATE privileges on the specified table, regardless of the operation actually performed.

- When inserting or updating data from another table using a SELECT query, you must have SELECT privilege on that table.

If you are the Owner (creator) of the table, you are automatically granted all privileges for that table. Otherwise, you must be granted privileges for the table. Failing to do so results in an SQLCODE –99 error. To determine if you have the appropriate privileges, use the %CHECKPRIV command. To assign table privileges, use the GRANT command. For more details, see Privileges.

More About

Uniqueness Checks

INSERT OR UPDATE determines if a row exists by matching UNIQUE column values to the existing data values. If a UNIQUE constraint violation occurs, **INSERT OR UPDATE** performs an update operation. The UNIQUE column value can be explicitly specified in **INSERT OR UPDATE** or it can be the result of a column default value or a computed value.

When **INSERT OR UPDATE** is run against a sharded table, if the shard key is the same as or a subset of the UNIQUE KEY constraint, **INSERT OR UPDATE** performs an update operation.

If the **INSERT OR UPDATE** attempts to perform an update because of any other unique values found (that are not the shard key), the command fails with an SQLCODE -120 error due to the unique constraint failure.

Counter Columns

When an **INSERT OR UPDATE** is executed, InterSystems IRIS initially assumes the operation is an insert. Therefore, it increments by 1 the internal counters used to supply integers to SERIAL (%Library.Counter) columns. An insert uses these incremented counter values to assign integer values to these columns. If, however, InterSystems IRIS determines that the operation needs to be an update, **INSERT OR UPDATE** has already incremented the internal counters, but it does not assign these incremented integer values to counter columns. If the next operation is an insert, this results in a gap in the integer sequence for these columns. This is shown in the following example:

1. The internal counter value is 4. **INSERT OR UPDATE** increments the internal counter, then inserts Row 5: internal counter = 5, SERIAL column value = 5.

2. **INSERT OR UPDATE** increments the internal counter, then determines that it must perform an update on an existing row: internal counter = 6, no change to column counters.

3. **INSERT OR UPDATE** increments internal counter, then inserts a row: internal counter = 7, SERIAL column value = 7.

IDENTITY and RowID Columns

The effect of **INSERT OR UPDATE** on the assignment of RowId values depends on whether an IDENTITY column is present:

- If no IDENTITY column is defined for the table, an insert operation causes InterSystems IRIS to automatically assign the next sequential integer value to the ID (RowID) column. Update operations have no effect on subsequent inserts. Thus, **INSERT OR UPDATE** performs the same insert operation as **INSERT**.

- If an IDENTITY column is defined for the table, an **INSERT OR UPDATE** causes InterSystems IRIS to increment by 1 the internal counter used to supply integers to the IDENTITY column, before determining if the operation will be an insert or an update. An insert operation assigns this incremented counter value to the IDENTITY column. If, however, InterSystems IRIS determines that the **INSERT OR UPDATE** operation needs to be an update, it has already incremented the internal counter, but it does not assign these incremented integer value. If the next **INSERT OR UPDATE** operation is an insert, this results in a gap in the integer sequence for the IDENTITY column. The RowID column value is taken from the IDENTITY column value, resulting in a gap in the assignment of ID (RowID) integer values.

IDKEY Column Values

When using **INSERT OR UPDATE**, you can only insert IDKEY column values, not update them. If the table has an IDKEY index and another UNIQUE constraint, **INSERT OR UPDATE** matches these columns to determine whether to perform an insert or an update. If the other key constraint fails, this forces **INSERT OR UPDATE** to perform an update rather than an insert. However, if the specified IDKEY column values do not match the existing IDKEY column values, this update fails and generates an SQLCODE -107 error, because the update is attempting to modify the IDKEY columns.

Consider a table with columns A, B, C, and D. The table has a primary key of (A,B) in an environment where the primary key is the IDKEY, and a UNIQUE constraint on columns (C,D).

SQL

```
SET OPTION PKEY_IS_IDKEY = TRUE
```

SQL

```
CREATE TABLE ABCD (
A INTEGER,
B INTEGER,
C INTEGER,
D INTEGER,
CONSTRAINT AB PRIMARY KEY (A,B),
CONSTRAINT CD UNIQUE (C,D))
```

The table also has two rows of data:

SQL

```
INSERT INTO ABCD SET A=1, B=1, C=2, D=2
```

SQL

```
INSERT INTO ABCD SET A=1, B=2, C=3, D=4
```

Suppose you try to insert this value:

SQL

```
INSERT OR UPDATE ABCD (A,B,C,D) VALUES (2,2,3,4)
```

Because the UNIQUE (C,D) constraint failed, this statement cannot perform an insert. Instead, it attempts to update Row 2. The IDKEY for Row 2 is (1,2), so the INSERT OR UPDATE statement attempts to change the column A value from 1 to 2. Because you cannot change an IDKEY value, the update fails with an SQLCODE -107 error.

Reset your environment to the default settings, where the primary key is not the IDKEY.

SQL

```
SET OPTION PKEY_IS_IDKEY = FALSE
```

See Also

- CREATE TABLE

- INSERT

- UPDATE

- Modifying the Database

- Defining Tables

- Defining Views

- Transaction Processing

- SQLCODE error messages

%INTRANSACTION (SQL)

Shows transaction state.

Synopsis

```
%INTRANSACTION
%INTRANS
```

Description

The **%INTRANSACTION** statement sets SQLCODE to indicate the transaction state:

- SQLCODE=0 if currently in a transaction.

- SQLCODE=100 if not in a transaction.

%INTRANSACTION returns SQLCODE=0 while a transaction is in progress. This transaction can be an SQL transaction initiated by **START TRANSACTION** or **SAVEPOINT**. It can also be an ObjectScript transaction initiated by TSTART.

Transaction nesting has no effect on **%INTRANSACTION**. **SET TRANSACTION** has no effect on **%INTRANSACTION**.

You can also determine transaction state using $TLEVEL. **%INTRANSACTION** only indicates whether a transaction is in progress. **$TLEVEL** indicates both whether a transaction is in progress and the current number of transaction levels.

Examples

The following embedded SQL example shows how **%INTRANSACTION** sets SQLCODE:

ObjectScript

```
 NEW SQLCODE
 &sql(%INTRANSACTION)
 WRITE "Before %INTRANS SQLCODE=",SQLCODE," TL=",$TLEVEL,!
 &sql(SET TRANSACTION %COMMITMODE EXPLICIT)
 NEW SQLCODE
 &sql(%INTRANSACTION)
 WRITE "SetTran %INTRANS SQLCODE=",SQLCODE," TL=",$TLEVEL,!
&sql(START TRANSACTION)
 NEW SQLCODE
 &sql(%INTRANSACTION)
 WRITE "StartTran %INTRANS SQLCODE=",SQLCODE," TL=",$TLEVEL,!
 &sql(SAVEPOINT a)
 NEW SQLCODE
 &sql(%INTRANSACTION)
 WRITE "Savepoint %INTRANS SQLCODE=",SQLCODE," TL=",$TLEVEL,!
 &sql(ROLLBACK TO SAVEPOINT a)
 NEW SQLCODE
 &sql(%INTRANSACTION)
 WRITE "Rollback to Savepoint %INTRANS SQLCODE=",SQLCODE," TL=",$TLEVEL,!
 &sql(COMMIT)
 NEW SQLCODE
 &sql(%INTRANSACTION)
 WRITE "After Commit %INTRANS SQLCODE=",SQLCODE," TL=",$TLEVEL
```

See Also

- COMMIT ROLLBACK SAVEPOINT SET TRANSACTION START TRANSACTION $TLEVEL

- Transaction Processing

JOIN (SQL)

A SELECT subclause that creates a table based on the data in two tables.

Synopsis

Inner Join

```
SELECT ... FROM table1 INNER JOIN table2 ON condition
SELECT ... FROM table1 INNER JOIN table2 USING (column, column2, ...)
SELECT ... FROM table1 JOIN table2 ...

SELECT ... FROM table1 NATURAL INNER JOIN table2
SELECT ... FROM table1 NATURAL JOIN table2
```

Left Outer Join

```
SELECT ... FROM table1 LEFT OUTER JOIN table2 ON condition
SELECT ... FROM table1 LEFT OUTER JOIN table2 USING (column, column2, ...)
SELECT ... FROM table1 LEFT JOIN table2 ...

SELECT ... FROM table1 NATURAL LEFT OUTER JOIN table2
SELECT ... FROM table1 NATURAL LEFT JOIN table2
```

Right Outer Join

```
SELECT ... FROM table1 RIGHT OUTER JOIN table2 ON condition
SELECT ... FROM table1 RIGHT OUTER JOIN table2 USING (column, column2, ...)
SELECT ... FROM table1 RIGHT JOIN table2 ...

SELECT ... FROM table1 NATURAL RIGHT OUTER JOIN table2
SELECT ... FROM table1 NATURAL RIGHT JOIN table2
```

Full Outer Join

```
SELECT ... FROM table1 FULL OUTER JOIN table2 ON condition
SELECT ... FROM table1 FULL JOIN table2 ON condition
```

Cross Join

```
SELECT ... FROM table1 CROSS JOIN table2
```

Description

The JOIN operation combines matching rows from two tables into a single table. Rows across two tables are considered a match when they have identical values in one or more specified columns. To further limit the returned rows in the joined table, you can specify additional restrictions.

Use joins to generate reports and queries that link related data across tables. Specify JOIN operations in a **SELECT** query as part of the FROM clause. Within a query, you can specify multiple inner and outer joins in any order.

Inner Join

An INNER JOIN returns the matching rows from the first and second table. For example:

SQL

```
SELECT Table1.Letter, Table2.Number
FROM Table1
INNER JOIN Table2
ON Table1.ID = Table2.ID
```
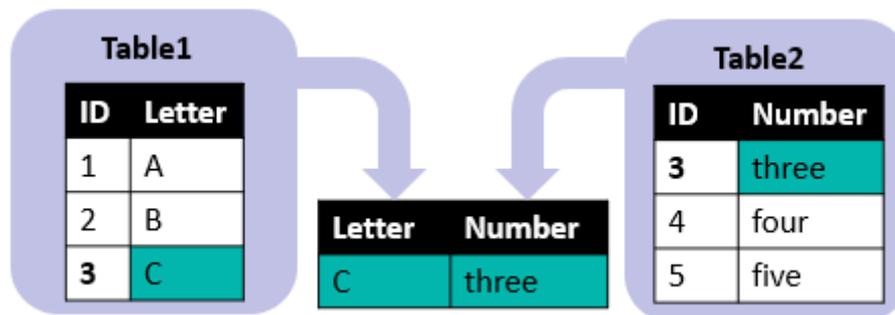
- **SELECT ... FROM** *table1* **INNER JOIN** *table2* **ON** *condition* returns the rows from *table1* and *table2* that match the condition expression specified in the ON clause. You can specify the ON clause anywhere within a join expression.

  This query returns the names of employees and their companies, joining data from the `Sample.Employee` table (aliased to `E`) and `Sample.Company` table (aliased to `C`). It returns `E.Name` and `C.Name` values only for rows in which the `CompanyID` column of both tables have matching values.

  SQL

  ```
  SELECT E.Name, C.Name
  FROM Sample.Employee AS E
  INNER JOIN Sample.Company AS C
  ON E.CompanyID = C.CompanyID
  ```

  This query additionally restricts the data by returning only rows of employees who are older than 20.

  SQL

  ```
  SELECT E.Name, C.Name
  FROM Sample.Employee AS E
  INNER JOIN Sample.Company AS C
  ON E.CompanyID = C.CompanyID AND E.Age > 20
  ```

  Example: Join Table Data Using Inner and Outer Joins

- **SELECT ... FROM** *table1* **INNER JOIN** *table2* **USING (***column*, *column2*, **...)** returns the rows from *table1* and *table2* that have matching values in the specified columns. The columns specified in the USING clause must appear in both tables. Use this syntax to express equality conditions more succinctly that the ON syntax, provided that the columns being linked have the same names in both tables. In multi-join queries, you can specify a USING clause only for the first join.

  This query performs the same join as in the previous syntax, because both columns have a `CompanyID` column that they can join on.

  SQL

  ```
  SELECT E.Name, C.Name
  FROM Sample.Employee AS E
  INNER JOIN Sample.Company AS C
  USING (CompanyID)
  ```

  Example: Join on Identically Named Columns Across Two Tables

- **SELECT ... FROM** *table1* **JOIN** *table2* **...** is equivalent to the previous INNER JOIN syntaxes.

- **SELECT ... FROM** *table1* **NATURAL INNER JOIN** *table2* performs an INNER JOIN on all identically named columns across the two tables. In multi-join queries, you can specify only one NATURAL join and it must be the first join.

This query performs the same operation as in the previous syntaxes, assuming that `CompanyID` is the only column that appears in both tables. If the tables include multiple identically named columns, then the query also joins on those columns before returning the matching results.

SQL

```
SELECT E.Name, C.Name
FROM Sample.Employee AS E
NATURAL INNER JOIN Sample.Company AS C
```

Example: Join on Identically Named Columns Across Two Tables

- **SELECT ... FROM** *table1* **NATURAL JOIN** *table2* is equivalent to the NATURAL INNER JOIN syntax.

Left Outer Join

A LEFT OUTER JOIN returns all rows from the first table and any rows from the second table that match rows from the first table. In the joined table, non-matching rows of columns from the second table are populated with null values. For example:

SQL

```
SELECT Table1.Letter, Table2.Number
FROM Table1
LEFT OUTER JOIN Table2
ON Table1.ID = Table2.ID
```
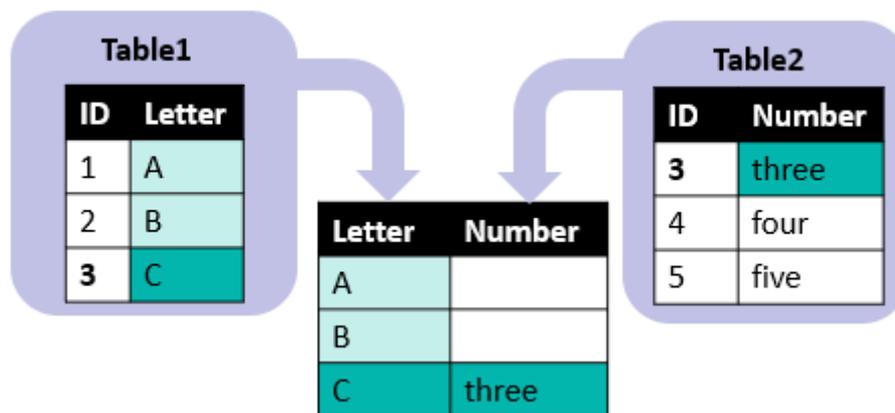

- **SELECT ... FROM** *table1* **LEFT OUTER JOIN** *table2* **ON** *condition* returns all rows from *table1* and joins them with any rows from *table2* that satisfy the condition expression specified in the ON clause.

This query returns the names of employees and their companies, joining data from the `Sample.Employee` table (aliased to `E`) and `Sample.Company` table (aliased to `C`). It returns all `E.Name` values but only `C.Name` values in rows where the `CompanyID` column of both tables have matching values. In non-matching rows, `C.Name` values are set to NULL.

SQL

```
SELECT E.Name, C.Name
FROM Sample.Employee AS E
LEFT OUTER JOIN Sample.Company AS C
ON E.CompanyID = C.CompanyID
```

Note: Alternatively, instead of using the explicit LEFT OUTER JOIN syntax, you can use the more succinct *implicit join* specified by the arrow syntax (–>) in the **SELECT** statement. For example, this query is equivalent to the previous query:

SQL

```
SELECT Name, CompanyID->Name
FROM Sample.Employee
```

This syntax assumes that the `CompanyID` column from `Sample.Employee` references the IDs of rows in the `Sample.Company` table, which contains the `Name` column that is being joined. For more details on working with implicit joins, see Implicit Joins.

Examples:

– Join Table Data Using Inner and Outer Joins

– Set Additional Restrictions on Joined Data

• **SELECT ... FROM** *table1* **LEFT OUTER JOIN** *table2* **USING (***column***, *column2*, ...)** returns all rows from *table1* and any rows from *table2* that have matching values in the specified columns. The columns must appear in both tables.

This query performs the same join as in the previous syntax, because both columns have a `CompanyID` column that they can join on.

SQL

```
SELECT E.Name, C.Name
FROM Sample.Employee AS E
LEFT OUTER JOIN Sample.Company AS C
USING (CompanyID)
```

Example: Join on Identically Named Columns Across Two Tables

• **SELECT ... FROM** *table1* **LEFT JOIN** *table2* **...** is equivalent to the LEFT OUTER JOIN syntaxes.

• **SELECT ... FROM** *table1* **NATURAL LEFT OUTER JOIN** *table2* performs a LEFT OUTER JOIN on all identically named columns across the two tables. If an expression contains multiple joins, specify the NATURAL join first. A NATURAL join does not merge columns that have the same name.

This query performs the same operation as in the previous syntaxes, assuming that `CompanyID` is the only column that appears in both tables. If the tables include multiple identically named columns, then the query performs an additional join per column.

SQL

```
SELECT E.Name, C.Name
FROM Sample.Employee AS E
NATURAL LEFT OUTER JOIN Sample.Company AS C
```

Example: Join on Identically Named Columns Across Two Tables

• **SELECT ... FROM** *table1* **NATURAL LEFT JOIN** *table2* is equivalent to the NATURAL LEFT OUTER JOIN syntax.

Right Outer Join

A RIGHT OUTER JOIN returns all rows from the second table and any rows from the first table that match rows from the second table. In the joined table, non-matching rows of columns from the first table are populated with null values. For example:

SQL

```
SELECT Table1.Letter, Table2.Number
FROM Table1
RIGHT OUTER JOIN Table2
ON Table1.ID = Table2.ID
```
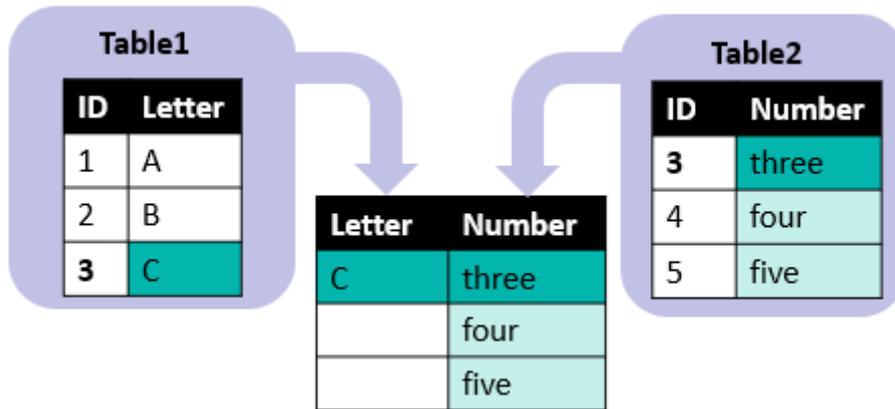

- **SELECT ... FROM *table1* RIGHT OUTER JOIN *table2* ON *condition*** returns all rows from *table2* and joins them with any rows from *table1* that satisfy the condition expression specified in the ON clause.

  This query returns the names of employees and their companies, joining data from the `Sample.Employee` table (aliased to `E`) and `Sample.Company` table (aliased to `C`). It returns all `C.Name` values but only `E.Name` values in rows where the `CompanyID` column of both tables have matching values. In non-matching rows, `E.Name` values are set to NULL.

  ### SQL

  ```
  SELECT E.Name, C.Name
  FROM Sample.Employee AS E
  LEFT OUTER JOIN Sample.Company AS C
  ON E.CompanyID = C.CompanyID
  ```

  Examples:

  - Join Table Data Using Inner and Outer Joins

  - Set Additional Restrictions on Joined Data

- **SELECT ... FROM *table1* RIGHT OUTER JOIN *table2* USING (*column*, *column2*, ...)** returns all rows from *table2* and any rows from *table1* that have matching values in the specified columns. The columns must appear in both tables.

  This query performs the same join as in the previous syntax, because both columns have a `CompanyID` column that they can join on.

  ### SQL

  ```
  SELECT E.Name, C.Name
  FROM Sample.Employee AS E
  RIGHT OUTER JOIN Sample.Company AS C
  USING (CompanyID)
  ```

  Example: Join on Identically Named Columns Across Two Tables

- **SELECT ... FROM *table1* RIGHT JOIN *table2* ...** is equivalent to the RIGHT OUTER JOIN syntaxes.

- **SELECT ... FROM *table1* NATURAL RIGHT OUTER JOIN *table2*** performs a RIGHT OUTER JOIN on all identically named columns across the two tables. If an expression contains multiple joins, specify the NATURAL join first. A NATURAL join does not merge columns that have the same name.

This query performs the same operation as in previous syntaxes, provided that `CompanyID` is the only column that appears in both tables. If the tables include multiple identically named columns, then the query performs one join per column.

SQL

```
SELECT E.Name, C.Name
FROM Sample.Employee AS E
NATURAL RIGHT OUTER JOIN Sample.Company AS C
```

Example: Join on Identically Named Columns Across Two Tables

- **SELECT ... FROM** *table1* **NATURAL RIGHT JOIN** *table2* is equivalent to the NATURAL RIGHT OUTER JOIN syntax.

Full Outer Join

A FULL OUTER JOIN joins all rows from both tables. In the joined table, non-matching rows of columns from either table are populated with null values. For example:

SQL

```
SELECT Table1.Letter, Table2.Number
FROM Table1
FULL OUTER JOIN Table2
ON Table1.ID = Table2.ID
```
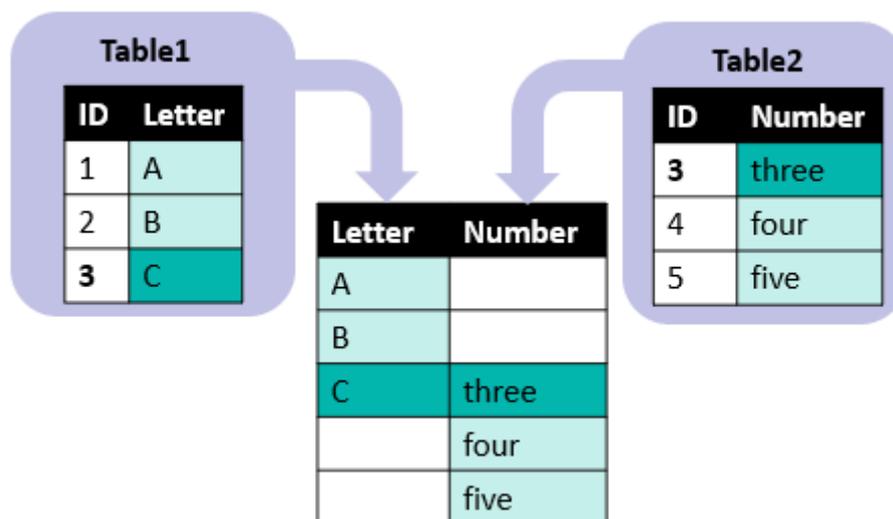

- **SELECT ... FROM** *table1* **FULL OUTER JOIN** *table2* **ON** *condition* returns all rows of *table1* and *table2* that match the specified *condition*.

  This query returns the names of people and the companies that they work for, joining data from `Sample.Person` and `Sample.Company`. For each row, if the person specified by `PersonID` is missing either `Company` or `Person` column data, that column value is NULL.

SQL

```
SELECT P.Name, E.Company
FROM Sample.Person AS P
INNER JOIN Sample.Employee AS E
ON P.PersonID = E.PersonID
```

Example: Join Table Data Using Inner and Outer Joins

- **SELECT ... FROM** *table1* **FULL JOIN** *table2* **ON** *condition* is equivalent to the FULL OUTER JOIN syntax.

Full outer joins do not support the USING or NATURAL syntaxes.

Cross Join

A CROSS JOIN crosses every row of the first table with every row of the second table. For example:

SQL

```
SELECT Table1.Letter, Table2.Number
FROM Table1
CROSS JOIN Table2
```
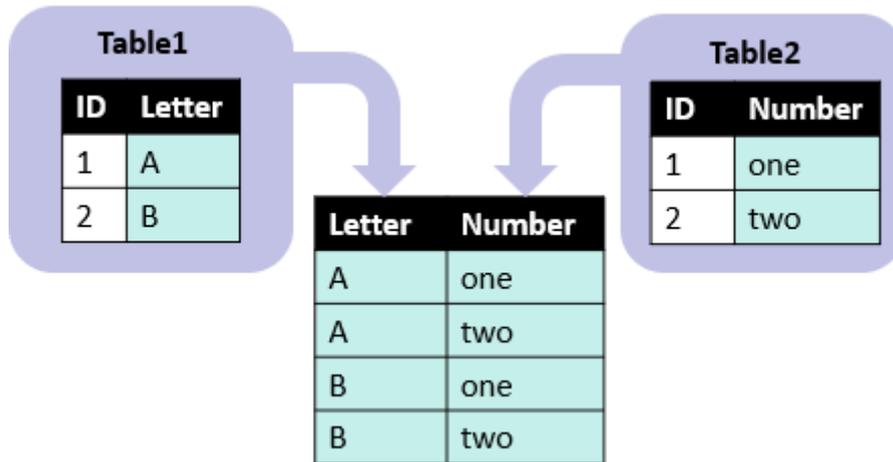

- **SELECT ... FROM** *table1* **CROSS JOIN** *table2* crosses every row of *table1* with every row of *table2*, resulting in a large, logically comprehensive table with much data duplication. Usually this join is performed by providing a comma-separated list of tables in the FROM clause, then using the WHERE clause to specify restrictive conditions.

  This query returns a row for each combination of rows in `Sample.LettersAtoZ` and `Sample.Numbers1to10`.

  ```
  SELECT * FROM Sample.LettersAtoZ CROSS JOIN Sample.Numbers1to10
  ```

  This query is equivalent to the previous query.

  SQL

  ```
  SELECT * FROM Sample.LettersAtoZ, Sample.Numbers1to10
  ```

Attempting to perform a cross join involving a local table and an external table linked through an ODBC or JDBC gateway connection (for example, `FROM Sample.Person, Mylink.Person`) results in an SQLCODE -161 error. To perform this cross join, you must specify the linked table as a subquery. For example: `FROM Sample.Person,(SELECT * FROM Mylink.Person)`.

The explicit use of the JOIN keyword has higher precedence than specifying a cross join using comma syntax. InterSystems IRIS® thus interprets `t1,t2 JOIN t3` as `t1,(t2 JOIN t3)`.

Arguments

table1, table2

Names of the tables being joined. Specify the first table, *table1*, after the FROM keyword. Specify the second table, *table2*, after the JOIN keyword.

- In a join with an ON clause, you can specify tables, views, or subqueries for either operand of the join.

- In a NATURAL or USING join, you can specify only simple base table references (not views or subqueries) for either operand of the join.

Both *table1* and *table2* support table aliases.

condition

One or more condition expression predicates, specified in the ON clause to restrict the rows being joined. JOIN supports most of the predicates supported by InterSystems SQL. However, you cannot use the **FOR SOME %ELEMENT** collection predicate to limit a join operation.

You can associate multiple condition expressions using AND, OR, and NOT logical operators. AND takes precedence over OR. To nest and group condition expressions, use parentheses. For example:

SQL

```
SELECT Patient.PName, Doctor.DName
FROM Patient
INNER JOIN Doctor
ON Patient.DocID = Doctor.DocID AND
   NOT (Doctor.State = 'NH' OR Doctor.State = 'MA')
```

condition has the following restrictions:

- *condition* can reference only tables explicitly specified in the ANSI keyword JOIN operation. Referencing tables specified in the FROM clause results in an SQLCODE -23 error.

- *condition* can reference only columns that are in the operands of the JOIN. Syntax precedence in multiple joins can cause the ON clause to fail. For example, this query fails because t1 and t3 are not operands of a join. t1 joins with the result set of t2 JOIN t3.

    SQL

    ```
    SELECT * FROM t1,t2 JOIN t3 ON t1.p1=t3.p3
    ```

    Either of the following changes in syntax result in the successful execution of this query:

    SQL

    ```
    SELECT * FROM t1 CROSS JOIN t2 JOIN t3 ON t1.p1=t3.p3
    ```

    SQL

    ```
    SELECT * FROM t2,t1 JOIN t3 ON t1.p1=t3.p3
    ```

- In OUTER JOIN clauses, if all the conditions affecting a table use comparisons that can pass null values, and that table is itself a target of an outer join, this can result in an SQLCODE -94 error. For example, this LEFT OUTER JOIN query is invalid:

    SQL

    ```
    SELECT * FROM Table1
    LEFT OUTER JOIN Table2 ON Table1.k = Table2.k
    LEFT OUTER JOIN Table3 ON COALESCE(Table1.k,Table2.k) = Table3.k
    ```

    Similar examples using FULL OUTER JOIN or RIGHT OUTER JOIN also have this restriction.

column

A column name, or comma-separated list of columns names, specified in the USING clause to join columns with the same names in both tables. Enclose the column list in parentheses. Only explicit column names are permitted. You cannot specify the `%ID` row that references the auto-generated RowID column. Duplicate column names are ignored. Columns with the same name are not merged.

Examples

Join Table Data Using Inner and Outer Joins

In this example, you create two sample tables, combine the data into one table using different INNER JOIN, LEFT OUTER JOIN, and RIGHT OUTER JOIN syntaxes, and compare the different joined results.

Create Tables

This examples uses two tables:

- `Sample.HighestPeaks` — Elevation (in feet) of mountains with the highest peaks, worldwide.

- `Sample.Himalayas` — Names of mountains in the Himalayas.

Although not specified in this example, assume that the `MountainID` and `PeakID` columns of both tables are foreign key references to a larger mountain database. Therefore, rows with the same ID column value in both tables refer to the same mountain.

Create the `Sample.HighestPeaks` table and insert three rows of data. Display the table.

SQL

```
CREATE TABLE Sample.HighestPeaks (
  PeakID INTEGER UNIQUE NOT NULL,
  Elevation INTEGER NOT NULL)
```

SQL

```
INSERT INTO Sample.HighestPeaks VALUES (1, 29032)
```

SQL

```
INSERT INTO Sample.HighestPeaks VALUES (2, 28251)
```

SQL

```
INSERT INTO Sample.HighestPeaks VALUES (3, 28169)
```

SQL

```
SELECT * FROM Sample.HighestPeaks
```

PeakID	Elevation
1	29032
2	28251
3	28169

Create the `Sample.Himalayas` table and insert three rows of data. The omitted `MountainID` of 2 is intentional. Assume that the mountain with an ID of 2 is not in the Himalayas. Display the table.

SQL

```
CREATE TABLE Sample.Himalayas (
  MountainID INTEGER UNIQUE NOT NULL,
  Name VARCHAR(30) UNIQUE NOT NULL)
```

SQL

```
INSERT INTO Sample.Himalayas VALUES (1, 'Everest')
```

SQL

```
INSERT INTO Sample.Himalayas VALUES (3, 'Kangchenjunga')
```

SQL

```
INSERT INTO Sample.Himalayas VALUES (4, 'Lhotse')
```

SQL

```
SELECT * FROM Sample.Himalayas
```

MountainID	Name
1	Everest
3	Kangchenjunga
4	Lhotse

Perform INNER JOIN

Combine the mountain name and elevation data from the two tables by using an INNER JOIN, joining them on the `MountainID` and `PeakID` columns. The joined table includes data only for the mountains with IDs of 1 and 3, because these IDs appear in both tables.

SQL

```
SELECT H.Name, P.Elevation
FROM Sample.Himalayas AS H
INNER JOIN Sample.HighestPeaks as P
ON H.MountainID = P.PeakID
```

Name	Elevation
Everest	29032
Kangchenjunga	28169

Perform LEFT OUTER JOIN

Combine the name and elevation data by using a LEFT OUTER JOIN, joining them on the `MountainID` and `PeakID` columns. The joined table includes all rows from the first table (`Sample.Himalayas`) but only the rows from the second table (`Sample.HighestPeaks`) with `PeakID` values of 1 and 3, which also appear in the `MountainID` column of the first table. The missing elevation of the mountain Lhotse takes a NULL value.

SQL

```
SELECT H.Name, P.Elevation
FROM Sample.Himalayas AS H
LEFT OUTER JOIN Sample.HighestPeaks as P
ON H.MountainID = P.PeakID
```

Name	Elevation
Everest	29032
Kangchenjunga	28169
Lhotse	

Perform RIGHT OUTER JOIN

Combine the name and elevation data by using a RIGHT OUTER JOIN, joining them on the `MountainID` and `PeakID` columns. The joined table includes all rows from the second table (`Sample.HighestPeaks`) but only the rows from the first table (`Sample.Himalayas`) with `MountainID` values of 1 and 3, which also appear in the `PeakID` column of the second table. The missing name of the mountain with an elevation of 28,251 feet akes a NULL value.

SQL

```
SELECT H.Name, P.Elevation
FROM Sample.Himalayas AS H
RIGHT OUTER JOIN Sample.HighestPeaks as P
ON H.MountainID = P.PeakID
```

Name	Elevation
Everest	29032
	28251
Kangchenjunga	28169

Perform FULL OUTER JOIN

Combine the name and elevation data by using a FULL OUTER JOIN, joining them on the `MountainID` and `PeakID` columns. The joined table includes all rows from both tables. The missing mountain names and elevations take NULL values.

SQL

```
SELECT H.Name, P.Elevation
FROM Sample.Himalayas AS H
FULL OUTER JOIN Sample.HighestPeaks as P
ON H.MountainID = P.PeakID
```

Name	Elevation
Everest	29032
Kangchenjunga	28169
Lhotse	
	28251

Delete Tables

Delete the sample tables when you are done.

SQL

```
DROP TABLE Sample.Himalayas
```

SQL

```
DROP TABLE Sample.HighestPeaks
```

Join on Identically Named Columns Across Two Tables

This example shows the different syntaxes you can use when joining columns that have identical names across the two tables.

Consider two tables:

*   `Patient` — Contains information about patients, including an ID code for the patient's primary doctor, DocID.

*   `Doctor` — Contains information about doctors, including their ID code, DocID.

This INNER JOIN returns the patient and doctor names.

SQL

```
SELECT Patient.PName, Doctor.DName
FROM Patient
INNER JOIN Doctor
ON Patient.DocID = Doctor.DocID
```

Because the joining columns have the same name in both tables (`DocID`), you can replace the ON clause with a USING clause. With this syntax, you specify only the column, in parentheses, and omit the table names.

SQL

```
SELECT Patient.PName, Doctor.DName
FROM Patient
INNER JOIN Doctor
USING (DocID)
```

You can also specify the USING clause with a LEFT OUTER JOIN or RIGHT OUTER JOIN, but the FULL OUTER JOIN is not supported.

SQL

```
SELECT Patient.PName, Doctor.DName
FROM Patient
RIGHT OUTER JOIN Doctor
USING (DocID)
```

SQL

```
SELECT Patient.PName, Doctor.DName
FROM Patient
LEFT OUTER JOIN Doctor
USING (DocID)
```

If `DocID` is the only identically named column across the two tables, then you can simplify further and use the NATURAL JOIN syntax.

SQL

```
SELECT Patient.PName, Doctor.DName
FROM Patient
NATURAL INNER JOIN Doctor
```

SQL

```
SELECT Patient.PName, Doctor.DName
FROM Patient
NATURAL LEFT OUTER JOIN Doctor
```

SQL

```
SELECT Patient.PName, Doctor.DName
FROM Patient
NATURAL RIGHT OUTER JOIN Doctor
```

If the two tables contain other identical columns, then NATURAL JOIN also links those columns in the join operation. For greater specificity over the columns being joined, use the USING or ON clauses. Full outer joins do not support NATURAL JOINs.

Set Additional Restrictions on Joined Data

This example shows how the setting of additional restrictions can affect the returned data from various joins.

Consider two tables:

- `Patient` — Contains information about patients, including an ID code for the patient's primary doctor, DocID.

- `Doctor` — Contains information about doctors, including their ID code, DocID.

This INNER JOIN returns the patient and doctor names of doctors who are over 45 years old.

SQL

```
SELECT Patient.PName, Doctor.DName
FROM Patient
INNER JOIN Doctor
ON Patient.DocID = Doctor.DocID AND Doctors.Age > 45
```

Performing a LEFT OUTER JOIN of the same query does not eliminate NULL values in the non-matching rows of the table being joined. For example, this LEFT OUTER JOIN still returns NULL values in the `Doctor.DName` column.

SQL

```
SELECT Patient.PName, Doctor.DName
FROM Patient
LEFT OUTER JOIN Doctor
ON Patient.DocID = Doctor.DocID AND Doctors.Age > 45
```

You can eliminate NULL values by moving the age condition into the WHERE clause, which processes after the join operation. However, this effectively converts the query into an INNER JOIN. For example, this query is equivalent to the first query in this example:

SQL

```
SELECT Patient.PName, Doctor.DName
FROM Patient
LEFT OUTER JOIN Doctor
ON Patient.DocID = Doctor.DocID
WHERE Doctors.Age > 45
```

Adding an IS NULL clause preserves the original LEFT OUTER JOIN behavior but is more verbose than the original LEFT OUTER JOIN query.

SQL

```
SELECT Patient.PName, Doctor.DName
FROM Patient
LEFT OUTER JOIN Doctor
ON Patient.DocID = Doctor.DocID
WHERE Doctors.Age > 45 AND Doctors.Age IS NULL
```

This behavior is similar for RIGHT OUTER JOIN operations. In a FULL OUTER JOIN, specifying conditions does not affect which rows are returned, because the operation returns all rows from both tables, regardless of matches.

Performance

Query Optimizer Effect on Joins

To maximize performance of join operations, the SQL optimizer might not join tables in the order in which they are specified. Instead, the optimizer determines the table join order based on statistics it gathers on the table, such as Tune Table.

In most cases, the SQL optimizer strategy provides optimal results. However, to override the default optimization strategy for a specific query, you can specify Query Optimization Options immediately after the **FROM** keyword.

- %INORDER, %FIRSTTABLE, and %STARTTABLE — For complex queries containing multiple joins, these options explicitly set the order in which to join tables. You cannot use these keywords with a cross join or a right outer join Attempting to do so results in an SQLCODE -34 error.

- %NOFLATTEN — This option disables subquery flattening, which converts certain subqueries to explicit joins. When the number of subqueries is small, subquery flattening can substantially improve join performance. As the number of subqueries increases, however, subquery flattening might start to degrade performance and might require disabling using this keyword.

ON Clause Indexing

Specifying indexes on columns referenced in the ON clause of a join can substantially improve query performance. An ON clause can use an existing index that satisfies only some of the join conditions. An ON clause specifying conditions on multiple columns can use an index containing only a subset of those columns as subscripts to partially satisfy the join. InterSystems IRIS tests the join condition on the remaining columns directly from the table.

The collation type of a field referenced in an ON clause should match the collation type that it has in the corresponding index. A collation type mismatch can cause an index to not be used. However, if a join condition is on a column with %EXACT collation, but only an index on the collated column value is available, InterSystems IRIS can use that index to limit the rows to be checked for the exact value. For more details on collation type matching, see Index Collation.

To disable an index for an ON clause condition, preface it with the %NOINDEX keyword. For more details on indexes and performance, see Using Indexes in Query Processing and Index Optimization Options.

Alternatives

InterSystems IRIS supports two formats for representing outer joins:

1. (Recommended) The ANSI standard syntax: LEFT OUTER JOIN and RIGHT OUTER JOIN. SQL Standard syntax puts the outer join in the FROM clause of the SELECT statement, rather than the WHERE clause, as shown in the following example:

   SQL

   ```
   SELECT table1.columnA, table2.columnB
   FROM table1
   LEFT OUTER JOIN table2
   ON (table1.columnX = table2.columnY)
   ```

2. The ODBC Specification outer join extension syntax, using the escape-syntax {oj *joinExpression* }, where *joinExpression* is any ANSI standard join syntax.

A join with an ON clause can use only the ANSI join keyword syntax.

See Also

- SELECT, FROM, ORDER BY, WHERE

- ALTER TABLE, CREATE TABLE, DROP TABLE

- **INSERT**, **UPDATE**
- **Defining Tables**
- **Querying the Database**
- **SQLCODE error messages**

LOAD DATA (SQL)

Loads data into a table.

Synopsis

Load from File

```
LOAD DATA FROM FILE filePath INTO table
LOAD DATA FROM FILE filePath INTO table (column, column2, ...)
LOAD DATA FROM FILE filePath COLUMNS (header type, header2 type2, ...) INTO table ...
LOAD DATA FROM FILE filePath COLUMNS (header type, header2 type2, ...) INTO table ... VALUES (header,
 header2, ...)
LOAD DATA FROM FILE filePath INTO table ... USING jsonOptions
```

Load from JDBC Source

```
LOAD DATA FROM JDBC CONNECTION jdbcConnection TABLE jdbcTable INTO table
LOAD DATA FROM JDBC CONNECTION jdbcConnection TABLE jdbcTable INTO table (column,column2, ...)
LOAD DATA FROM JDBC CONNECTION jdbcConnection TABLE jdbcTable INTO table ... VALUES (header,header2
...)
LOAD DATA FROM JDBC URL path TABLE jdbcTable ...
```

Bulk Loading Options

```
LOAD BULK DATA FROM ...
LOAD %NOJOURN DATA FROM ...
LOAD BULK %NOJOURN DATA FROM ...
```

Description

The **LOAD DATA** command loads data from a source into a previously defined InterSystems IRIS® SQL table. The source can be a data file or a table accessed using JDBC. Use this command for the rapid population of a table with well-validated data.

If the table you are loading data into is empty, **LOAD DATA** populates the table with the source data rows. If the table already contains data, **LOAD DATA** inserts the source data rows into the table without overwriting any existing rows.

When you load data, the *%ROWCOUNT* variable indicates the number of rows successfully loaded. If a row in the input data contains an error, **LOAD DATA** skips loading this row and proceeds with loading the next row. SQLCODE does not report this as an error, but the `%SQL_Diag.Result` log indicates how many records failed to load. For more details, see View Diagnostic Logs of Loaded Data.

Note: The **LOAD DATA** command uses an underlying Java-based engine that requires a Java Virtual Machine (JVM) installation on your server. If you already have a JVM set up and accessible in your PATH environment variable, then the first time you use **LOAD DATA**, InterSystems IRIS automatically uses that JVM to start an External Language Server. To customize your External Language Server to use a specific JVM, or to use a remote server, see Managing External Server Connections.

Load from File

Load from File Without Headers

Use these syntaxes if your source file does not contain a header row in the first line of the file. Otherwise, **LOAD DATA** loads the header row into the table.

- **LOAD DATA FROM FILE** *filePath* **INTO** *table* loads the source data from the file specified by *filePath* into the target SQL table. By default, **LOAD DATA** matches the columns from the data source to the target table by position. **LOAD DATA** uses the SQL column order (SELECT * column order).

- If the data source has more columns than the input table, then the excess columns are ignored and not loaded into the table.

- If the data source has fewer columns the input table, none of the data is loaded into the table.

The **LOAD DATA** command expects that the data type of the loaded data matches the data type of the target table columns.

This statement loads all the columns from the `countries` CSV source file into the columns that are in the corresponding positions of the target `Sample.Countries` table.

```
LOAD DATA FROM FILE 'C://mydata/countries.csv'
INTO Sample.Countries
```

- **LOAD DATA FROM FILE** *filePath* **... INTO** *table* (*column*, *column2*, *...*) loads source data positionally for only the specified target table columns. If the data source has fewer columns than the input table, then those columns are empty in the inserted rows.

This statement loads the first three columns from the `countries` CSV file into the `Name`, `Continent`, and `Region` columns of the `Sample.Countries` table. Even if the table stores these columns in a different order, or if there are columns in between the three shown here, **LOAD DATA** still loads data only into `Name`, `Continent`, and `Region`.

```
LOAD DATA FROM FILE 'C://mydata/countries.csv'
INTO Sample.Countries (Name,Continent,Region)
```

- **LOAD DATA FROM FILE** *filePath* **COLUMNS** (*header type*, *header2 type2*, *...*) **INTO** *table* enables you to load data from source files that have a different column order than the target table. The `COLUMNS` clause provides header names and data types for the columns in the source files. The header names must match the names of columns in the target table and the data type must be consistent with the data types of those table columns.

  - If the `INTO` *table* clause specifies target columns, then the columns named in the `COLUMNS` clause must also appear in the `INTO` *table* clause, but they can be in any order.

  - If the `INTO` *table* clause does not specify target columns, then **LOAD DATA** loads the source columns positionally into the table. The `COLUMNS` clause must name all columns that appear in the target table.

This statement loads three columns from the `countries` CSV file into corresponding columns in the `Sample.Countries` table. If the `Sample.Countries` table has a different column order than the source file (for example, `Name`, `SurfaceArea`, `Continent` instead of `Name`, `Continent`, `SurfaceArea`), the table column order does not change.

```
LOAD DATA FROM FILE 'C://mydata/countries.csv'
COLUMNS (
    Name VARCHAR(50),
    Continent VARCHAR(30),
    SurfaceArea Integer)
INTO Sample.Countries (Name,Continent,SurfaceArea)
```

- **LOAD DATA FROM FILE** *filePath* **COLUMNS** (*header type*, *header2 type2*, *...*) **INTO** *table* **... VALUES** (*header*, *header2*, *...*) additionally enables you to load a subset of columns from the source file into the target columns. These column names do not need to match the target table column names.

The `VALUES` clause specifies the source columns, as named by the headers in the `COLUMNS` clause, to load into the target table.

  - If the `INTO` *table* clause specifies target columns, then **LOAD DATA** loads the source columns into the target columns in the order those columns are specified. The number of source column headers in `VALUES` must match the number of columns in the INTO table clause.

  - If the `INTO` *table* clause does not specify target columns, then **LOAD DATA** loads the source columns positionally into the table. The number of source headers in **VALUES** must match the number of columns in the table.

This statement loads three columns from the `countries` CSV file into corresponding columns in the `Sample.Countries` table. The `COLUMNS` clause includes a header name for an additional column, `src_continent`, that is not loaded into the table. This column name is ignored, but it must be included so that **LOAD DATA** can load the data from the subsequent columns (`src_region` and `src_surface_area`) into the table.

```
LOAD DATA FROM FILE 'C://mydata/countries.csv'
COLUMNS (
    src_name VARCHAR(50),
    src_continent VARCHAR(30),
    src_region VARCHAR(30),
    src_surface_area INTEGER)
INTO Sample.Countries (Name,SurfaceArea,Region)
VALUES (src_name,src_surface_area,src_region)
```

If you specify the `VALUES` clause without the `COLUMNS` clause, then the `VALUES` clause is ignored.

Load from File with Headers and Specify Options

Use this syntax if the first line of your source file contains a header row. Using this syntax, you can specify an option to skip the header row. Other options include changing the default column, skipping additional rows beyond the header, and changing the default escape character.

• **LOAD DATA FROM FILE** *filePath* **INTO** *table* **... USING** *jsonOptions* specifies loading options by using a JSON object or a string containing a JSON object.

This statement uses a JSON object to specify that the file contains a header row, so that **LOAD DATA** does not include this row in the table. In this statement, it is assumed that the header names in the `countries` CSV file match the header names of the `Sample.Countries` table columns.

```
LOAD DATA FROM FILE 'C://mydata/countries.csv'
INTO Sample.Countries
USING {"from":{"file":{"header":true}}}
```

Note: If the header text does not validate against the field data type, such as an integer field with a header named `"Total"`, **LOAD DATA** might omit the header row anyway. However, this method of validation rejection is unreliable and is not recommend. Omit the header with a `USING` clause instead.

The statement loads data from three columns in the `countries` CSV file into the three corresponding columns in the `Sample.Countries` table. In this statement, the header names in the `countries` CSV file do not match the header names of the `Sample.Countries` table columns. The `VALUES` clause specifies the column header names obtained from the file. Data from these columns is then loaded into the table columns that are in the corresponding position of the `INTO` *table* clause.

```
LOAD DATA FROM FILE 'C://mydata/countries.csv'
INTO Sample.Countries (Name,Region,SurfaceArea)
VALUES (country_name,region_name,surface_area)
USING {"from":{"file":{"header":true}}}
```

Load from JDBC Source

• **LOAD DATA FROM JDBC CONNECTION** *connection* **TABLE** *jdbcTable* **INTO** *table* loads data from an external JDBC data source into the target table. The data source, *jdbcTable*, is a JDBC-compliant SQL table that you connect to by using a defined SQL Gateway Connection, *connection*. For more details, see Connecting the SQL Gateway via JDBC.

This statement loads all columns from the JDBC source table, `countries`, into the corresponding columns of the `Sample.Countries` table.

```
LOAD DATA FROM JDBC CONNECTION MyJDBCConnection
TABLE countries
INTO Sample.Countries
```

- **LOAD DATA FROM JDBC CONNECTION** *connection* **TABLE** *jdbcTable* (*column*, *column2*, ...) loads JDBC source data positionally for only the specified target table columns. If the JDBC source has fewer columns than the input table, then those columns are empty in the inserted rows.

  This statement loads the first three columns from the JDBC `countries` table into the `Name`, `Continent`, and `Region` columns of the `Countries` table. Even if the table stores these columns in a different order, or if there are columns in between the three shown here, **LOAD DATA** still loads data only into `Name`, `Continent`, and `Region`.

  ```
  LOAD DATA FROM JDBC CONNECTION MyConnection
  TABLE countries
  INTO Sample.Countries (Name,Continent,Region)
  ```

- **LOAD DATA FROM JDBC CONNECTION** *connection* **TABLE** *jdbcTable* ... **INTO** *table* ... **VALUES** (*header,header2* ...) loads data form the JDBC source for only the columns that have the header names specified in the `VALUES` clause. Using this syntax, you can place column data at any position in the JDBC source table into columns at any position in the target table.

  The number of columns in the `INTO` table clause must match the number of headers in the `VALUES` clause. If the `INTO` table clause does not specify any columns, then the number of headers in the `VALUES` clause must match the number of the headers in the table. In this case, the source data is loaded positionally into the table.

  This statement loads data from the `name`, `surface_area`, and `region` columns of the JDBC `countries` table into the corresponding columns of the `Sample.Countries` table.

  ```
  LOAD DATA FROM JDBC CONNECTION MyConnection
  TABLE countries
  INTO Sample.Countries (Name,SurfaceArea,Region)
  VALUES (name,surface_area,region)
  ```

  The way the VALUES clause matches SQL column names positionally is similar to INSERT command syntax.

- **LOAD DATA FROM JDBC URL** path **TABLE** *jdbcTable* **INTO** *table* loads data from an external JDBC data source into the target table. The data source, defined by *path*, is located at the connection URL for the data source. For more details, see Connecting the SQL Gateway via JDBC.

  This statement loads all columns from the JDBC source table, `countries`, into the corresponding columns of the `Sample.Countries` table.

  ```
  LOAD DATA FROM JDBC URL jdbc:oracle:thin:@//oraserver:1521/SID
  TABLE countries
  INTO Sample.Countries
  ```

Bulk Loading Options

These options disable common checks and operations performed when data is inserted into a table. Disabling these options can significantly speed up the loading of data with a large number of rows.

CAUTION: These bulk loading options can result in the loading of invalid data. Before using these options, make sure that the data is valid and is from a reliable source.

- **LOAD BULK DATA FROM ...** loads data with these **INSERT** %keyword options specified:

  – %NOCHECK — Disables unique value checking, foreign key referential integrity checking, NOT NULL constraints (required field checks), and validation for column data types, maximum column lengths, and column data constraints.

  – %NOINDEX — Disables setting of index maps during **INSERT** processing. During the **LOAD BULK DATA** operation, SQL statements run against the target table might be incomplete or return incorrect results.

  – %NOLOCK — Disables locking of the row upon **INSERT**.

To use the BULK keyword, you must have %NOCHECK, %NOINDEX, and %NOLOCK administrative privileges, which you can set by using the GRANT command.

This statement loads bulk data from a file.

```
LOAD BULK DATA FROM FILE 'C://mydata/countries.csv'
INTO Sample.Countries
```

- **LOAD %NOJOURN DATA FROM ...** loads data with the %NOJOURN **INSERT** %keyword option specified, which suppresses journaling and disables transactions for the duration of the insert operations. To use the %NOJOURN option, you must have %NOJOURN SQL administrative privileges, which you can set by using the GRANT command.

  This form of the LOAD DATA command acquires a table-level lock on the target table, but each row is inserted with %NOLOCK. The table level lock is released when the command completes.

  This statement loads data from a table over a JDBC connection and disables journaling.

```
LOAD %NOJOURN DATA FROM JDBC CONNECTION MyJDBCConnection
TABLE countries
INTO Sample.Countries
```

- **LOAD %NOJOURN BULK DATA FROM ...** loads data with the **INSERT** %keyword options from the previous syntaxes specified. You can specify %NOJOURN and BULK in either order.

Arguments

filePath

The server-side location of a text file containing the data to load, specified as a complete file path enclosed in quotes.

- Each line in the file specifies a separate row to be loaded into the table. Blank lines are ignored.

- Data values in a row are separated by a column separator character. Comma is the default column separator character. All data fields must be indicated by column separators, including unspecified data indicated by placeholder column separators. You can define a different column separator character by specifying the `columnseparator` option in the `USING` *jsonOptions* clause.

- By default, no escape character is defined. To include the column separator character as a literal in a data value, enclose the data value with quotation marks. To include a quotation mark in a quoted data value, double the quote character (""). You can define an escape character specifying the `escapechar` option in the `USING` *jsonOptions* clause.

- By default, data values are specified in the order of the fields in the table (or view). You can use the `COLUMNS` clause to specify the data in a different order. You can use a view to load a data record to a table by supplying only values for the fields that are defined in the view.

- All data in a data file record is validated against the table's data criteria, including the number of data fields in the record, and the data type and data length of each field. A data file record that fails validation is passed over (not loaded). No error message is issued. Data loading continues with the next record.

Note: Date or timestamp data should be written in ODBC timestamp format (`'yyyy-mm-dd hh:mm:ss'`) to ensure validation.

table

The table to load the data into. A table name can be qualified (schema.tablename), or unqualified (tablename). An unqualified table name takes the default schema name. You can specify a view to load data in the table accessed through the view.

column

The table columns to load file data into, specified in the order of the columns in the file. This list of column names enables you to specify selected table columns and to match the order of the data file items to the columns in *table*. Unspecified columns that are defined in the table take their default values. If this clause is omitted, all user-defined fields in *table* must be represented in the data file.

header

A comma-separated list of header values used to identify columns to load from the data source.

- When loading data from a file source that does not contain a header row, specify headers in the COLUMNS *header type*, *header2 type2*, ...) clause to name the columns.

  - If you include a VALUES clause, specify these header names in VALUES (*header*, *header2*) to select which columns to load into the table.

  - If you do not include a VALUES clause, then these header names must match the column names in the target table.

- When loading data from a file source that contains a header row, specify headers in the VALUES (*header*, *header2*) clause to identify which headers in the source file to load data from. These header names must exist in the source file.

- When loading data from a JDBC source, specify headers in the VALUES (*header*, *header2*) clause to identify which columns in the JDBC source table to load data from. These header names must exist in the JDBC source table.

type

The data type of the headers specified in the COLUMNS *header type*, *header2 type2*, ...) clause. The data type for each column must be compatible with the table's data type. The table's data length, not the COLUMNS data length, is used to validate the data.

jsonOptions

Loading options, specified in the USING clause as a JSON (JavaScript Object Notation) object or a string containing a JSON object. These syntaxes are equivalent:

```
USING {"from":{"file":{"header":true}}}
```

```
USING '{"from":{"file":{"header":true}}}'
```

Use these JSON objects to set loading options that cannot be set using SQL keywords. Specify these objects using nested key:value pair syntax, as described in JSON Values.

The primary use of this object is to set options of the loaded data that supplements the FROM FILE syntax, although there are options for parallelizing and permitting errors during the execution of the **LOAD DATA**. This example shows a sample JSON object with multiple options specified. The whitespace shown here is optional and is provided for readability only.

```
USING
{
  "from": {
    "file": {
        "header": true,
        "skip": 2
        "charset": "UTF-8"
        "escapechar": "\\"
        "columnseparator": "\t"
    }
  }
}
```

This table shows the options that you can specify. Unspecified options use the default values.

Option	Description	Example

Option	Description	Example
`from.file.header`	Set to `true` (1) to indicate that the first line of the source file is a header row. Column names in this header can then be specified and used in a `VALUES` clause, if no `COLUMNS` clause is specified. For more details, see Load from File with Headers and Specify Options. Default: `false` (0)	`{"from":{"file":{"header":true}}}`
`from.file.skip`	Specify the number of lines at the start of the file to skip. If `header` is set to true, then skip indicates the number of lines to skip in addition to the header. Default: `0`	`{"from":{"file":{"skip":2}}}`
`from.file.charset`	Specify the character set used to parse input data. Default: **LOAD DATA** uses the character set of the host operating system.	`{"from":{"file":{"charset":"UTF-8"}}}`
`from.file.escapechar`	Specify the escape character used for literal values, such as column separator characters that are used within a column value. Default: None	`{"from":{"file":{"escapechar":"\\"}}}`
`from.file.columnseparator`	Specify the column separator character. Default: `","`	`{"from":{"file":{"columnseparator":";"}}}`

Option	Description	Example
`into.jdbc.threads`	Specify the number of threads to parallelize the JDBC writer across. This option may be used even when not loading data from a JDBC source. Each thread feeds a single server process performing **INSERT** commands. If it is important that data is loaded into the table in the exact order it is defined in the table, you should specify `"threads":1.` In general, you should specify a lower value when multiple **LOAD DATA** commands run in parallel. Default: $System.Util.NumberOfC-PUs() - 2	`{"into":{"jdbc":{"threads":4}}}`
`maxerrors`	The maximum number of errors that may arise during the **LOAD DATA** command before the entire operation is determined to be a failure, closing a transaction and rolling back all changes. Default: 0	`{"maxerrors":5}`

jdbcConnection

A defined SQL Gateway Connection name used to load data from a JDBC source. For details on establishing a JDBC connection, see Connecting the SQL Gateway via JDBC.

jdbcTable

The external SQL data source table accessed over a JDBC connection. For details on establishing a JDBC connection, see Connecting the SQL Gateway via JDBC.

path

The SQL Gateway Connection URL used to load data from a JDBC source. For details on establishing a JDBC connection, see Connecting the SQL Gateway via JDBC.

Security and Privileges

LOAD DATA is a privileged operation that requires a user to have INSERT privileges to modify the table you are loading into and to access the JVM on your server.

INSERT Privileges

To execute **LOAD DATA** on a table, the user must have table-level or column-level privileges for that table. In particular, the user must have INSERT privilege on the table. The Owner (creator) of the table is automatically granted all privileges for that table. If you are not the Owner, you must be granted privileges for the table. If you do not have the proper privileges, InterSystems IRIS raises a SQLCODE -99 error.

Table-level privileges are equivalent, but not identical to, having column-level privileges on all columns of the table. If you only have column-level privileges on a subset of the table's columns, you will only be able to load data into those columns. If you attempt to load data into a column that you do not have permissions for, InterSystems IRIS raises a SQLCODE -99 error.

To determine if you have the appropriate privileges, use %CHECKPRIV. To assign table privileges to a user, use GRANT. For more details, see Privileges.

Gateway Privileges

To execute **LOAD DATA**, the user must have access to the JVM on your server. As with access to any external language server in InterSystems IRIS, such a connection is privileged. Users need the %Gateway_Object:USE privilege to appropriately access the JVM.

Transaction Considerations

Atomicity

LOAD DATA is an atomic operation. Like other atomic operations, a **LOAD DATA** command is completely rolled back if it is not successful by making use of transactions by default; if the command cannot be completed, no data is inserted and the database reverts to its state before issuing the **LOAD DATA**. Unlike other atomic operations, there are notable exceptions to this rule. These exceptions are as follows:

- **LOAD BULK DATA** and **LOAD %NOJOURN DATA** do not start transactions.

- **LOAD DATA** is unique among atomic operations because it enables the use of *jsonoption* in a **USING** clause. With the *maxerrors* JSON option, you can specify an upper limit on insertion errors during a **LOAD DATA** command. If this limit is reached, the transaction will fail and the database reverts to its state before issuing the **LOAD DATA**. If the limit is not reached, the transaction succeeds and the successfully loaded data will appear in the database; however, data that failed to be loaded in will not appear in the database.

You can modify this default for the current process within SQL by invoking SET TRANSACTION %COMMITMODE. You can modify this default for the current process in ObjectScript by invoking the **SetOption()** method, using this syntax:

```
SET status=$SYSTEM.SQL.Util.SetOption("AutoCommit",intval,.oldval)
```

The following *intval* integer options are available:

- 1 or IMPLICIT (autocommit on — default) — Calling **LOAD DATA** initiates and completes its own transaction.

- 2 or EXPLICIT (autocommit off) — If no transaction is in progress, **LOAD DATA** automatically initiates a transaction, but you must explicitly **COMMIT** or **ROLLBACK** to end the transaction. In EXPLICIT mode, the number of database operations per transaction is user-defined.

- 0 or NONE (no auto transaction) — No transaction is initiated when you invoke **LOAD DATA**. A failed **LOAD DATA** operation can leave the database in an inconsistent state, with some rows inserted and some not inserted. To provide transaction support in this mode, you must use **START TRANSACTION** to initiate the transaction and **COMMIT** or **ROLLBACK** to end the transaction.

    A sharded table is always in no auto-transaction mode, which means all inserts, updates, and deletes to sharded tables are performed outside the scope of a transaction.

To determine the atomicity setting for the current process, use the **GetOption("AutoCommit")** method, as shown in this ObjectScript example:

ObjectScript

```
SET stat=$SYSTEM.SQL.Util.SetOption("AutoCommit",$RANDOM(3),.oldval)
IF stat'=1 {WRITE "SetOption failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET x=$SYSTEM.SQL.Util.GetOption("AutoCommit")
  IF x=1 {
    WRITE "Default atomicity behavior",!
    WRITE "automatic commit or rollback" }
  ELSEIF x=0 {
    WRITE "No transaction initiated, no atomicity:",!
    WRITE "failed DELETE can leave database inconsistent",!
    WRITE "rollback is not supported" }
  ELSE { WRITE "Explicit commit or rollback required" }
```

Examples

Load Data From CSV File into SQL Tables and Views

This example shows how to load data stored in a comma-separated value (CSV) file into an existing table and view.

Create the table to load data into. This table contains three fields specifying membership data: a member ID, the membership term length in months, and the US state where the member lives, using the two-character state abbreviation.

SQL

```
CREATE TABLE Sample.Members (
    MemberId INT PRIMARY KEY,
    MemberTerm INT DEFAULT 12,
    MemberState CHAR(2))
```

Copy these data records into a text file. Save the file on your local machine and name it members.csv. This file specifies membership IDs and member state values. The second row is missing a value, as indicated by a placeholder comma inserted before where the value would be.

```
6138830,MA
1720936,
4293608,NH
```

Use **LOAD DATA** to load the data into the Sample.Members table. Replace the path shown here with the path where you saved the file.

```
LOAD DATA FROM FILE 'C://temp/members.csv' INTO Sample.Members (MemberId,MemberState)
```

Examine the data. The MemberId and MemberState columns have been populated. The source file did not contain data for the MemberTerm column, so these column values default to 12. The missing row value is loaded in as a NULL value.

SQL

```
SELECT * FROM Sample.Members
```

MemberId	MemberTerm	MemberState
6138830	12	MA
1720936	12	
4293608	12	NH

LOAD DATA does not report an SQLCODE error for the missing data because the overall SQLCODE result of **LOAD DATA** is 0 (success). A **LOAD DATA** operation is considered successful if:

• **LOAD DATA** can access the source.

• The target table exists.

- The **LOAD DATA** operation is valid. For example, the operation specifies the correct number of columns and the column names exist in the target table.

To view the SQLCODE errors for individual rows, along with other information about the **LOAD DATA** operation, you can use the %SQL_Diag.Result and %SQL_Diag.Message tables. For more details, see View Diagnostic Logs of Loaded Data.

View the messages from the most recent **LOAD DATA** operation. The row with the missing state abbreviation reports an SQLCODE error of -104. The results shown are truncated for readability.

SQL

```
SELECT actor,message,severity,sqlcode
FROM %SQL_Diag.Message
WHERE diagResult =
  (SELECT TOP 1 resultId
  FROM %SQL_Diag.Result
  ORDER BY resultId DESC)
```

actor	message	severity	sqlcode
server	{"resultid":"1","bufferrowcount":500, ... }	info	0
FileReader	Reader Complete: Total Input file read time: 23 ms,	completed	0
JdbcWriter	[SQLCODE: <-104>:<Field validation failed in INSERT>] [%msg: ... (Varchar Value: 'state...' Length: 5) > maxlen: (2)>]	error	-104
JdbcWriter	Writer Complete: Total write time: 72 ms,	completed	0

If you create a view from a table, you can also load data into the table by using the view. Create a view that shows only the membership ID and state columns of the `Sample.Members` table.

SQL

```
CREATE VIEW Sample.VMem (MId,State) AS SELECT MemberId,MemberState FROM Sample.Members
```

Copy these additional data records into a text file. Save the file on your local machine and name it `members2.csv`.

```
6785674,VT
4564563,RI
4346756,ME
```

Use **LOAD DATA** to load this new CSV data into the table by using the view you created.

```
LOAD DATA FROM FILE 'C://temp/members2.csv' INTO Sample.VMem(MId,State)
```

View the data returned by the view, which includes the data from both loaded CSV files.

SQL

```
SELECT * FROM Sample.VMem
```

MId	State
6138830	MA
1720936	
4293608	NH
6785674	VT
4564563	RI
4346756	ME

View the data in the base table, which includes combined column data from both CSV files. The default value of 12 is applied to the values in the MemberTerm column for the second loaded CSV file as well.

SQL

```
SELECT * FROM Sample.Members
```

MemberId	MemberTerm	MemberState
6138830	12	MA
1720936	12	
4293608	12	NH
6785674	12	VT
4564563	12	RI
4346756	12	ME

Delete the view and table.

SQL

```
DROP VIEW Sample.VMem
```

SQL

```
DROP TABLE Sample.Members
```

Troubleshooting

View Diagnostic Logs of Loaded Data

Each call to **LOAD DATA** generates a new row in the %SQL_Diag.Result table. This table contains diagnostic information about the operation. You can view these rows by using a **SELECT** query. For example, this query returns the five most recent **LOAD DATA** calls.

SQL

```
SELECT TOP 5 * FROM %SQL_Diag.Result ORDER BY createTime DESC
```

The returned table has these columns:

- `ID` — Integer ID of the log entry. This value is the primary key of the table.

- `resultId` — Same as ID.

- `createTime` — Timestamp for when the **LOAD DATA** operation took place and the log entry row was created. Timestamps are in UTC (Coordinated Universal Time), not local time.

- `namespace` — Namespace in which the **LOAD DATA** operation took place.

- `processId` — Integer ID of the process that performed the **LOAD DATA** operation.

- `user` — User who performed the **LOAD DATA** operation.

- `sqlCode` — SQLCODE of the overall **LOAD DATA** operation.

- `inputRecordCount` — Number of records successfully loaded.

- `errorCount` — The number of errors that occurred. Includes errors that cause **LOAD DATA** command and failures to load or write individual rows of data.

- `maxErrorCount` — Maximum number of row insertion errors that **LOAD DATA** tolerates before failing the operation.

- `status` — Status of the **LOAD DATA** operation. While **LOAD DATA** is executing, the status is set to `"In Progress"`. When the **LOAD DATA** operation is complete, the status is updated to `"Complete"`. If the **LOAD DATA** execution produces an error, the status is updated to `"Failed"`.

- `statement` — Text of the SQL statement that was executed to produce this `%SQL_Diag.Result` record.

The **%SQL_Diag.Message** table provides detailed message data for each **LOAD DATA** operation logged in the **%SQL_Diag.Result** table. The `diagResult` column of **%SQL_Diag.Message** is a foreign key reference to the `resultId` column of **%SQL_Diag.Result** table, enabling you to access messages for individuals **LOAD DATA** operations.

For example, this query returns all error messages associated with the **LOAD DATA** operation that has a `resultId` of 29. You can use this data to diagnose which rows of the table failed to load.

SQL

```
SELECT *
FROM %SQL_Diag.Message
WHERE severity = 'error'
AND diagResult = 29
```

The returned table has these columns:

- `ID` — Integer ID of the message. This value is the primary key of the table.

- `actor` — Entity that reported the message, such as `server`, `FileReader`, or `JdbcWriter`.

- `diagResult` — Row ID of the **LOAD DATA** log entry recorded in the **%SQL_Diag.Result** table.

- `message` — Message data. For errors, this column includes SQLCODE values and %msg text.

- `messageTime` — Timestamp of the message in UTC (Coordinated Universal Time), not local time.

- `severity` — Level of severity of the message. severity is a logical integer that has a correspond display. Valid values are `"completed"`, `"info"`, `"warning"`, `"error"`, and `"abort"`.

- `sqlcode` — SQLCODE of the message. Messages with a severity of `"completed"` or `"info"` report an SQLCODE of 0. Messages with a severity of `"warning"` or `"error"` report the SQLCODE associated with that message. Messages with a severity of `"abort"` report an SQLCODE of -400.

See Also

- INSERT

- CREATE TABLE

- Importing and Exporting SQL Data

- SQLCODE error messages

LOCK (SQL)

Locks a table.

Synopsis

```
LOCK [TABLE] tablename IN EXCLUSIVE MODE
[WAIT seconds]  LOCK [TABLE] tablename IN SHARE MODE [WAIT seconds]
```

Description

LOCK and **LOCK TABLE** are synonymous.

The **LOCK** command explicitly locks an SQL table. This table must be an existing table for which you have the necessary privileges. If *tablename* is a nonexistent table, **LOCK** fails with a compile error. If *tablename* is a temporary table, the command completes successfully, but performs no operation. If *tablename* is a view, the command fails with an SQLCODE -400 error.

The **UNLOCK** command reverses the **LOCK** operation. An explicit **LOCK** remains in effect until you issue an explicit **UNLOCK** for the same mode, or until the process terminates.

You can use **LOCK** to lock a table multiple times; you must explicitly **UNLOCK** the table as many times as it was explicitly locked. Each **UNLOCK** must specify the same mode as the corresponding **LOCK**.

Privileges

The **LOCK** command is a privileged operation. Prior to using **LOCK** IN SHARE MODE it is necessary for your process to have SELECT privilege for the specified table. Prior to using **LOCK** IN EXCLUSIVE MODE it is necessary for your process to have INSERT, UPDATE, or DELETE privilege for the specified table. For IN EXCLUSIVE MODE, the INSERT or UPDATE privilege must be on at least one field of the table. Failing to hold sufficient privileges results in an SQLCODE -99 error (Privilege Violation). You can determine if the current user has the necessary privileges by invoking the %CHECKPRIV command. You can determine if a specified user has the necessary privileges by invoking the **$SYSTEM.SQL.Security.CheckPrivilege()** method. For privilege assignment, refer to the GRANT command.

These privileges are required to acquire the lock; they do not define the nature of the lock. An IN EXCLUSIVE MODE lock prevents other processes from performing INSERT, UPDATE, or DELETE operations, regardless of whether the lock holder has the corresponding privilege.

LOCK Modes

LOCK supports two modes: SHARE and EXCLUSIVE. These lock modes are independent of each other. You can apply both a SHARE lock and an EXCLUSIVE lock to the same table. A lock in EXCLUSIVE mode can only be unlocked by an **UNLOCK** in EXCLUSIVE mode. A lock in SHARE mode can only be unlocked by an **UNLOCK** in SHARE mode.

- `LOCK mytable IN SHARE MODE` prevents other processes from issuing an EXCLUSIVE lock on mytable, or invoking a DDL operation, such as **DROP TABLE**.

- `LOCK mytable IN EXCLUSIVE MODE` prevents other processes from issuing an EXCLUSIVE lock or a SHARE lock on mytable, performing an insert, update, or delete operation, or invoking a DDL operation, such as **DROP TABLE**.

LOCK permits read access to the table. Neither **LOCK** mode prevents other processes from performing a **SELECT** on the table in READ UNCOMMITTED mode (the default **SELECT** mode).

Locking Conflicts

- If a table is already locked by another user IN EXCLUSIVE MODE, you cannot lock it in any mode.

- If a table is already locked by another user IN SHARE MODE, you can also lock the table IN SHARE MODE, but you cannot lock it IN EXCLUSIVE MODE.

These **LOCK** conflicts generate an SQLCODE -110 error and generates a %msg such as the following: `Unable to acquire shared table-level lock for table 'Sample.Person'.`

Lock Timeout

LOCK attempts to acquire the specified SQL table lock until timeout occurs. When timeout occurs, **LOCK** generates an SQLCODE -110 error.

- If you have specified WAIT *seconds*, SQL table lock timeout occurs when that number of seconds elapses.

- Otherwise, SQL table lock timeout occurs when the current process SQL timeout elapses. You can set the lock timeout for the current process using the ProcessLockTimeout option of the **$SYSTEM.SQL.Util.SetOption()** method. You can also set the lock timeout for the current process using the SQL command SET OPTION with the LOCK_TIMEOUT option. (**SET OPTION** cannot be used from the SQL Shell.) The current process SQL lock timeout defaults to the system-wide SQL lock timeout.

- Otherwise, SQL table lock timeout occurs when the system-wide SQL timeout elapses. The system-wide default is 10 seconds. You can set the system-wide lock timeout in two ways:

  - Using the LockTimeout option of the **$SYSTEM.SQL.Util.SetOption()** method. This immediately changes the system-wide lock timeout default for new processes, and also resets the ProcessLockTimeout for the current process to this new system-wide value. Setting the system-wide lock timeout has no effect on the ProcessLockTimeout setting for other currently running processes.

  - Using the Management Portal, select **System Administration**, **Configuration**, **SQL and Object Settings**, **SQL**. View and edit the current setting of **Lock timeout (seconds)**. This changes the system-wide lock timeout default for new processes that start after you save the configuration change. It has no effect on currently running processes.

To return the current system-wide lock timeout value call the **$SYSTEM.SQL.Util.GetOption("LockTimeout")** method.

To return the lock timeout value for the current process call the **$SYSTEM.SQL.Util.GetOption("ProcessLockTimeout")** method.

Transaction Processing

A **LOCK** operation is not part of a transaction. Rolling back a transaction in which a **LOCK** is issued does not release the lock. An **UNLOCK** can be defined as occurring at the conclusion of the current transaction, or occurring immediately.

Other Locking Operations

Many DDL operations, including **ALTER TABLE** and **DELETE TABLE**, acquire an exclusive table lock.

The **INSERT**, **UPDATE**, and **DELETE** commands also perform locking. By default they lock at the record level for the duration of the current transaction; if one of these commands locks a sufficiently large number of records (1000 is the default setting), the lock is automatically elevated to a table lock. The **LOCK** command allows you to explicitly set a table level lock, giving you greater control over the locking of data resources. An **INSERT**, **UPDATE**, or **DELETE** can override a **LOCK** by specifying the %NOLOCK keyword.

The InterSystems SQL SET OPTION with the LOCK_TIMEOUT option sets the timeout for the current process for an **INSERT**, **UPDATE**, **DELETE**, or **SELECT** operation.

InterSystems SQL supports the CachedQueryLockTimeout option of the **$SYSTEM.SQL.Util.SetOption()** method.

Arguments

tablename

The name of the table to be locked. *tablename* must be an existing table. A *tablename* can be qualified (schema.table), or unqualified (table). An unqualified table name takes the default schema name. A schema search path is ignored.

IN EXCLUSIVE MODE/IN SHARE MODE

The IN EXCLUSIVE MODE keyword phrase creates a regular InterSystems IRIS lock. The IN SHARE MODE keyword phrase creates a shared InterSystems IRIS lock.

WAIT seconds

An optional integer specifying the number of seconds to attempt to acquire the lock before timing out. If omitted, the system default timeout is applied.

Examples

The following examples create a table and then lock it:

SQL

```
CREATE TABLE mytest (
ID NUMBER(12,0) NOT NULL,
CREATE_DATE DATE DEFAULT CURRENT_TIMESTAMP(2),
WORK_START DATE DEFAULT SYSDATE)
```

SQL

```
LOCK mytest IN EXCLUSIVE MODE WAIT 4
```

SQL programs run from the Management Portal spawn a process that terminates as soon as the program executes. Thus a lock is almost immediately released. Therefore, to observe a lock conflict, first issue a **LOCK mytest IN EXCLUSIVE MODE** command from a Terminal running the SQL Shell in the same namespace. Then run the above code locking program. Issue an **UNLOCK mytest IN EXCLUSIVE MODE** from the Terminal SQL Shell. Then rerun the above locking program.

See Also

- UNLOCK
- INSERT, UPDATE, DELETE
- SQL and Object Settings Pages
- SQLCODE error messages

OPEN (SQL)

Opens a cursor.

Synopsis

```
OPEN cursor-name
```

Description

An **OPEN** statement opens a cursor according to the parameters specified in the cursor's DECLARE statement. Once opened, a cursor can be fetched. An open cursor must be closed.

- Attempting to open a cursor that is not declared results in an SQLCODE -52 error.

- Attempting to open a cursor that is already open results in an SQLCODE -101 error.

- Attempting to fetch or close a cursor that is not open results in an SQLCODE -102 error.

A successful **OPEN** sets SQLCODE = 0, even if the result set is empty.

As an SQL statement, this is only supported from embedded SQL. Equivalent operations are supported through ODBC using the ODBC API.

Arguments

cursor-name

The name of the cursor, which has already been declared. The cursor name was specified in the **DECLARE** statement. Cursor names are case-sensitive.

Example

The following embedded SQL example shows a cursor (named EmpCursor) being opened and closed:

ObjectScript

```
SET name="LastName,FirstName",state="##"
&sql(DECLARE EmpCursor CURSOR FOR
     SELECT Name, Home_State
     INTO :name,:state FROM Sample.Person
     WHERE Home_State %STARTSWITH 'A')
WRITE !,"BEFORE: Name=",name," State=",state
&sql(OPEN EmpCursor)
IF SQLCODE '= 0 { WRITE "Open error: ",SQLCODE
                  QUIT }
NEW %ROWCOUNT,%ROWID
FOR { &sql(FETCH EmpCursor)
     QUIT:SQLCODE
     WRITE !,"DURING: Name=",name," State=",state }
WRITE !,"FETCH status SQLCODE=",SQLCODE
WRITE !,"Number of rows fetched=",%ROWCOUNT
&sql(CLOSE EmpCursor)
WRITE !,"AFTER: Name=",name," State=",state
```

See Also

- CLOSE, DECLARE, FETCH

- SQL Cursors

- SQLCODE error messages

PURGE CACHED QUERIES (SQL)

Deletes one or more cached queries.

Synopsis

```
PURGE [CACHED] QUERIES

PURGE [CACHED] QUERIES BY AGE n

PURGE [CACHED] QUERIES BY TABLE table-name

PURGE [CACHED] QUERIES BY NAME class-name [, class-name]
```

Description

The **PURGE CACHED QUERIES** command removes defined cached queries within a specified scope:

- **PURGE CACHED QUERIES** purges all cached queries in the current namespace.

- **PURGE CACHED QUERIES BY AGE *n*** purges all cached queries in the current namespace that have not been used (prepared) within the last *n* days. Specifying an *n* value of 0 purges all cached queries in the current namespace.

- **PURGE CACHED QUERIES BY TABLE *table-name*** purges all cached queries that reference a specified table. If a query references several tables, a single cached query is generated and listed for each of these tables. Issuing a **PURGE CACHED QUERIES BY TABLE** for any one of these tables purges this cached query from all of these tables.

- **PURGE [CACHED] QUERIES BY NAME *class-name*** purges cached queries specified by cached query class name. You can specify multiple cached queries as a comma-separated list. Listed cached queries can reference different tables, but all must be within the current namespace. Cashed query names are case-sensitive.

The CACHED keyword is optional.

If the specified *class-name* does not exist, or the letter case specified is not correct, that class name is skipped and the command proceeds to purge the next cached query in the list; for an invalid class name no operation is performed and no error is generated. If the specified table does not have any associated cached queries, or the table does not exist, no operation is performed and no error is generated.

Arguments

n

An integer number of days since the cached query was last used, specified as a quoted string.

table-name

The name of an existing table for which there are cached queries. A *table-name* can be qualified (schema.table), or unqualified (table). An unqualified table name takes the default schema name.

class-name

A cached query class name or a comma-separated list of cached query class names. Cached query class names are case-sensitive.

Examples

The following example purges the cached query specified by name:

```
PURGE CACHED QUERIES BY NAME %sqlcq.USER.cls2
```

The following example purges all cached queries that have not been used in the last two days:

```
PURGE CACHED QUERIES BY AGE "2"
```

See Also

- %SYSTEM.SQL.PurgeCQClass() purges a cached queries when given the name of a cached query class. This function should be used to purge individual cached queries.

REVOKE (SQL)

Removes privileges from a user or role.

Synopsis

Revoking Privileges

```
REVOKE admin-privilege FROM grantee
```

Revoking Roles

```
REVOKE role FROM grantee
```

Revoking Object Privileges

```
REVOKE [GRANT OPTION FOR] object-privilege
     ON object-list
     FROM grantee
     [CASCADE | RESTRICT] [AS grantor]

REVOKE [GRANT OPTION FOR]
     SELECT ON CUBE[S] object-list
     FROM grantee
```

Revoking Table-level/Column-level Privileges

```
REVOKE column-privilege (column-list)
     ON table
     FROM grantee
    [CASCADE | RESTRICT]
```

Description

The **REVOKE** statement revokes privileges that allow a user or role to perform specified tasks on specified tables, views, columns, or other entities. **REVOKE** can also revoke a role assignment from a user. **REVOKE** reverses the actions of the GRANT command; see that command for more details on privileges generally.

To revoke a privilege you must either:

- Be the user who granted the privilege

- Revoke the privilege through a CASCADE operation.

- Revoke a privilege granted by another user if you are logged in as a user with either the %Admin_Secure administrative resource with USE permission, or full security privileges on the system.

You can revoke a role or privilege from a specified user, a list of users, or all users (using the * syntax).

Because **REVOKE** prepares and executes quickly, and is generally run only once, InterSystems IRIS does not create a cached query for **REVOKE** in ODBC, JDBC, or Dynamic SQL.

A **REVOKE** completes successfully, even if no actual revoke can be performed (for example, the specified privilege was never granted or has already been revoked). However, if an error occurs during the **REVOKE** operation, SQLCODE is set to a negative number.

Revoking Roles

Roles can be granted or revoked via either the SQL **GRANT** and **REVOKE** commands, or via ^SECURITY InterSystems IRIS System Security. You can use **REVOKE** to revoke a role from a user or to revoke a role from another role. You cannot use InterSystems IRIS System Security to grant or revoke roles to other roles. The **$ROLES** special variable does not display roles granted to roles.

REVOKE can specify a single role, or a comma-separated list of roles to revoke. **REVOKE** can revoke one or more roles from a specified user (or role), a list of users (or roles), or all users (using the * syntax).

The **GRANT** command can grant a non-existent role to a user. You can use **REVOKE** to revoke a non-existent role from an existing user. However, the role name must be specified using the same letter case that was used to grant the role.

If you attempt to revoke an existing role from a non-existent user or role, InterSystems IRIS issues an SQLCODE -118 error. If you are not the SuperUser, and you attempt to revoke a role that you don't own and don't have ADMIN OPTION for, InterSystems IRIS issues an SQLCODE -112 error.

Revoking Object Privileges

Object privileges give a user or role some right to a particular object. You revoke an *object-privilege* ON an *object-list* FROM a *grantee*. An *object-list* can specify one or more tables, views, stored procedures, or cubes in the current namespace. By using comma-separated lists, a single **REVOKE** statement can revoke multiple object privileges on multiple objects from multiple users and/or roles.

You can use the asterisk (*) wildcard as the *object-list* value to revoke the *object-privilege* from all of the objects in the current namespace. For example, `REVOKE SELECT ON * FROM Deborah` revokes this user's SELECT privilege for all tables and views. `REVOKE EXECUTE ON * FROM Deborah` revokes this user's EXECUTE privilege for all non-hidden Stored Procedures.

You can use SCHEMA *schema-name* as the *object-list* value to revoke the *object-privilege* for all of the tables, views, and stored procedures in the named schema, in the current namespace. For example, `REVOKE SELECT ON SCHEMA Sample FROM Deborah` revokes this user's SELECT privilege for all objects in the Sample schema. You can specify multiple schemas as a comma-separated list; for example, `REVOKE SELECT ON SCHEMA Sample,Cinema FROM Deborah` revokes SELECT privilege for all objects in both the Sample and the Cinema schemas.

You can revoke an object privilege from a user or from a role. If you revoke it from a role, a user that only had that privilege through the role no longer has the privilege. A user that no longer has a privilege can no longer execute an existing cached query that requires that object privilege.

When **REVOKE** revokes an object privilege, it completes successfully and sets SQLCODE to 0. If **REVOKE** does not perform an actual revoke (for example, the specified object privilege was never granted or has already been revoked), it completes successfully and sets SQLCODE to 100 (no more data). If an error occurs during the **REVOKE** operation, it sets SQLCODE to a negative number.

Cubes are SQL identifiers that are not qualified by a schema name. To specify a cubes *object-list*, you must specify the CUBE (or CUBES) keyword. Because cubes can only have SELECT privilege, you can only revoke SELECT privilege from a cube.

Object privileges can be revoked by any of the following:

- The **REVOKE** command.

- The **$SYSTEM.SQL.Security.RevokePrivilege()** method.

- Via InterSystems IRIS System Security. Go to the Management Portal, select **System Administration**, **Security**, **Users** (or **System Administration**, **Security**, **Roles**) select **Edit** for the desired user or role, then select the **SQL Tables** or **SQL Views** tab. Select the desired **Namespace** from the drop-down list. Scroll down to the desired table, then click **revoke** to revoke privileges.

You can determine if the current user has a specified object privilege by invoking the %CHECKPRIV command. You can determine if a specified user has a specified table-level object privilege by invoking the **$SYSTEM.SQL.Security.CheckPrivilege()** method.

Revoking Object Owner Privileges

If you revoke the privileges on an SQL object from the owner of the object, the owner will still implicitly have privileges on the object. In order to completely revoke all privileges on the object from the owner of the object, the object must be changed to specify a different owner or no owner.

Revoking Table-level and Column-level Privileges

REVOKE can be used to reverse the granting of table-level privileges or column-level privileges. A table-level privilege provides access to all of the columns in a table. A column-level privilege provides access to every specified column in the table. Granting a column-level privilege to all of the columns in a table is functionally equivalent to granting a table-level privilege. However, the two are not functionally identical. A column-level **REVOKE** can only revoke privileges granted at the column level. You cannot grant a table-level privilege to the table, then revoke this privilege at the column level for one or more columns. In this case, the **REVOKE** statement has no effect on granted privileges.

CASCADE or RESTRICT

InterSystems IRIS supports the optional CASCADE and RESTRICT keywords to specify **REVOKE** *object-privilege* behavior. If neither keyword is specified, the default is RESTRICT.

You can use CASCADE or RESTRICT to specify whether revoking an *object-privilege* or *column-privilege* from a user will also revoke that privilege from any other users that received it via the WITH GRANT OPTION. CASCADE revokes all such associated privileges. RESTRICT (the default) causes **REVOKE** to fail when an associated privilege is detected. Instead it sets the SQLCODE -126 error "REVOKE with RESTRICT failed".

The use of these keywords is shown by the following example:

SQL

```
--UserA
   GRANT Select ON MyTable TO UserB WITH GRANT OPTION
```

SQL

```
--UserB
   GRANT Select ON MyTable TO UserC
```

SQL

```
--UserA
   REVOKE Select ON MyTable FROM UserB
   -- This REVOKE fails with SQLCODE -126
```

SQL

```
--UserA
   REVOKE Select ON MyTable FROM UserB CASCADE
   -- This REVOKE succeeds
   -- It revokes this privilege from UserB and UserC
```

Note that CASCADE and RESTRICT have no effect on a view created by UserB that references MyTable.

Effect on Cached Queries

When you revoke a privilege or role, InterSystems IRIS updates all cached queries on the system to reflect this change in privileges. However, when a namespace is inaccessible — for example, when an ECP connection to a database server is down — the **REVOKE** successfully completes but performs no operation on cached queries in that namespace. This is because **REVOKE** cannot update the cached queries in the unreachable namespace to revoke the privileges at the cached query level. No error is issued.

If the database server later comes up, the privileges for the cached queries in that namespace may be incorrect. It is advised that you purge cached queries in a namespace if a role or privilege might have been revoked while the namespace was not accessible.

InterSystems IRIS Security

The **REVOKE** command is a privileged operation. Prior to using **REVOKE** in embedded SQL, you must either be the grantor of the privilege or be logged in as a user with either the %Admin_Secure administrative resource with USE permission, or full security privileges on the system. Failing to do so results in an SQLCODE -99 error (Privilege Violation).

Use the **$SYSTEM.Security.Login()** method to assign a user with appropriate privileges:

ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql(        )
```

You must have the **%Service_Login:Use** privilege to invoke the **$SYSTEM.Security.Login** method. For further information, refer to %SYSTEM.Security in the *InterSystems Class Reference*.

Arguments

admin-privilege

An administrative-level privilege or a comma-separated list of administrative-level privileges previously granted to be revoked. The available *syspriv* options include sixteen object definition privileges and four data modification privileges.

The object definition privileges are: %CREATE_FUNCTION, %DROP_FUNCTION, %CREATE_METHOD, %DROP_METHOD, %CREATE_PROCEDURE, %DROP_PROCEDURE, %CREATE_QUERY, %DROP_QUERY, %CREATE_TABLE, %ALTER_TABLE, %DROP_TABLE, %CREATE_VIEW, %ALTER_VIEW, %DROP_VIEW, %CREATE_TRIGGER, %DROP_TRIGGER. Alternatively, you can specify %DB_OBJECT_DEFINITION, which revokes all 16 object definition privileges.

The data modification privileges are the %NOCHECK, %NOINDEX, %NOLOCK, %NOTRIGGER privileges for INSERT, UPDATE, and DELETE operations.

grantee

A list of one or more users having SQL System Privileges, SQL Object Privileges, or Roles. Valid values are a comma-separated list of users or roles, or "*". The asterisk (*) specifies all currently defined users who do not have the %All role.

AS grantor

This clause permits you to revoke a privilege granted by another user by specifying the name of the original grantor. Valid *grantor* values are a user name, a comma-separated list of user names, or "*". The asterisk (*) specifies all currently defined users who are grantors. To use the AS *grantor* clause, you must have the %All role or the %Admin_Secure resource.

role

A role or comma-separated list of roles whose privileges are being revoked from a user.

object-privilege

A basic-level privilege or comma-separated list of basic-level privileges previously granted to be revoked. The list may consist of one or more of the following: %ALTER, DELETE, SELECT, INSERT, UPDATE, EXECUTE, and REFERENCES. To revoke all privileges, use either "ALL [PRIVILEGES]" or "*" as the value for this argument. Note that you can only revoke SELECT privilege from cubes, because this is the only grantable cubes privilege.

obejct-list

A comma-separated list of one or more tables, views, stored procedures, or cubes for which the *object-privilege*(s) are being revoked. You can use the SCHEMA keyword to specify revoking the *object-privilege* from all objects in the specified schema. You can use "*" to specify revoking the *object-privilege* from all objects in the current namespace.

column-privilege

A basic-level privilege being revoked from one or more *column-list* listed columns. Available options are SELECT, INSERT, UPDATE, and REFERENCES.

column-list

A list of one or more column names, separated by commas and enclosed in parentheses.

table

The name of the table or view that contains the *column-list* columns.

Examples

The following embedded SQL example creates two users, creates a role, and assigns the role to the users. It then revokes the role from all users using the asterisk (*) syntax. If the user or the role already exists, the CREATE statement issues an SQLCODE -118 error. If the user does not exist, the GRANT or REVOKE statement issues an SQLCODE -118 error. If the user exists but the role does not, the GRANT or REVOKE statement issues SQLCODE 100. If the user and role exist, the GRANT or REVOKE statement issues SQLCODE 0. This is true even when the granting or revoking of the role has already been done, or if you are attempting to revoke a role that was never granted.

ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql(CREATE USER User1 IDENTIFY BY fredpw)
&sql(CREATE USER User2 IDENTIFY BY barneypw)
WRITE !,"CREATE USER error code: ",SQLCODE
&sql(CREATE ROLE workerbee)
WRITE !,"CREATE ROLE error code: ",SQLCODE
&sql(GRANT workerbee TO User1,User2)
WRITE !,"GRANT role error code: ",SQLCODE
&sql(REVOKE workerbee FROM *)
WRITE !,"REVOKE role error code: ",SQLCODE
```

In the following example, one user (Joe) grants a privilege and a different user (John) revokes that privilege, using the AS *grantor* clause:

SQL

```
/* User Joe */
GRANT SELECT ON Sample.Person TO Michael
```

SQL

```
/* User John */
REVOKE SELECT ON Sample.Person FROM Michael AS Joe
```

Note that John must have the %All role or the %Admin_Secure resource.

See Also

- SQL statements: CREATE USER, DROP USER, CREATE ROLE, DROP ROLE, GRANT, %CHECKPRIV

- SQL Users, Roles, and Privileges

- SQLCODE error messages

- ObjectScript: $ROLES and $USERNAME special variables

ROLLBACK (SQL)

Rolls back a transaction.

Synopsis

```
ROLLBACK [WORK]
ROLLBACK TO SAVEPOINT pointname
```

Description

A **ROLLBACK** statement rolls back a transaction, undoing work performed but not committed, decrementing the $TLEVEL transaction level counter, and releasing locks. **ROLLBACK** is used to restore the database to a previous consistent state.

- A **ROLLBACK** rolls back all work completed during the current transaction, resets the **$TLEVEL** transaction level counter to zero and releases all locks. This restores the database to its state before the beginning of the transaction. **ROLLBACK** and **ROLLBACK WORK** are equivalent statements; both versions are supported for compatibility.

- A **ROLLBACK TO SAVEPOINT** *pointname* rolls back all work done since the specified savepoint and decrements the **$TLEVEL** transaction level counter by the number of savepoints undone. When all savepoints have been either rolled back or committed and the transaction level counter reset to zero, the transaction is completed. If the specified savepoint does not exist, or has already been rolled back, **ROLLBACK** issues an SQLCODE -375 error and rolls back the entire current transaction.

A **ROLLBACK TO SAVEPOINT** must specify a *pointname*. Failing to do so results in an SQLCODE -301 error.

For details on establishing savepoints, refer to SAVEPOINT.

An SQLCODE -400 error is issued if a transaction operation fails to complete successfully.

Not Rolled Back

The following items are not affected by a **ROLLBACK** operation:

- A roll back does not decrement the IDKey counter for a default class. The IDKey is automatically generated by $INCREMENT (or $SEQUENCE), which maintains a count independent of the SQL transaction.

- A roll back does not reverse the creation, modification, or purging of a cached query. These operations are not treated as part of a transaction.

- A DDL operation or a Tune Table operation that occur within a transaction may create and run a temporary routine. This temporary routine is treated the same as a Cached Query. That is, the creation, compilation, and deletion of a temporary routine are not treated as part of the transaction. The execution of the temporary routine is considered part of the transaction.

For non-SQL items rolled back or not rolled back, refer to the ObjectScript TROLLBACK command.

Rollback Logging

Messages indicating that a rollback occurred, and errors encountered during the rollback operation are logged in the messages.log file in the MGR directory. You can use the Management Portal **System Operation**, **System Logs**, **Messages Log** option to view messages.log.

Transactions Suspended

The **TransactionsSuspended()** method of the %SYSTEM.Process class can be used to suspend and resume all current transactions for a process. Suspending transactions suspends journaling of changes. Therefore, if transaction suspension occurred during the current transaction, **ROLLBACK** cannot roll back any changes made while transactions were suspended;

however, **ROLLBACK** rolls back any changes made during the current transaction that occurred before or after the transaction suspension was in effect.

For further details, refer to Using ObjectScript for Transaction Processing.

ObjectScript Transaction Commands

ObjectScript and SQL transaction commands are fully compatible and interchangeable, with the following exception:

ObjectScript **TSTART** and SQL **START TRANSACTION** both start a transaction if no transaction is current. However, **START TRANSACTION** does not support nested transactions. Therefore, if you need (or may need) nested transactions, it is preferable to start the transaction with **TSTART**. If you need compatibility with the SQL standard, use **START TRANSACTION**.

ObjectScript transaction processing provides limited support for nested transactions. SQL transaction processing supplies support for savepoints within transactions.

Examples

The following Embedded SQL example demonstrates how a **ROLLBACK** restores the transaction level counter (**$TLEVEL**) to 0, the level immediately prior to the **START TRANSACTION**:

ObjectScript

```
&sql(SET TRANSACTION %COMMITMODE EXPLICIT)
  WRITE !,"Set transaction mode, SQLCODE=",SQLCODE
  WRITE !,"Transaction level=",$TLEVEL
&sql(START TRANSACTION)
  WRITE !,"Start transaction, SQLCODE=",SQLCODE
  WRITE !,"Transaction level=",$TLEVEL
&sql(SAVEPOINT a)
  WRITE !,"Set Savepoint a, SQLCODE=",SQLCODE
  WRITE !,"Transaction level=",$TLEVEL
&sql(SAVEPOINT b)
  WRITE !,"Set Savepoint b, SQLCODE=",SQLCODE
  WRITE !,"Transaction level=",$TLEVEL
  &sql(SAVEPOINT c)
  WRITE !,"Set Savepoint c, SQLCODE=",SQLCODE
  WRITE !,"Transaction level=",$TLEVEL
&sql(ROLLBACK)
  WRITE !,"Rollback transaction, SQLCODE=",SQLCODE
  WRITE !,"Transaction level=",$TLEVEL
```

The following Embedded SQL example demonstrates how a **ROLLBACK TO SAVEPOINT name** restores the transaction level (**$TLEVEL**) to the level immediately prior to the specified **SAVEPOINT**:

ObjectScript

```
&sql(SET TRANSACTION %COMMITMODE EXPLICIT)
  WRITE !,"Set transaction mode, SQLCODE=",SQLCODE
  WRITE !,"Transaction level=",$TLEVEL
&sql(START TRANSACTION)
  WRITE !,"Start transaction, SQLCODE=",SQLCODE
  WRITE !,"Transaction level=",$TLEVEL
&sql(SAVEPOINT a)
  WRITE !,"Set Savepoint a, SQLCODE=",SQLCODE
  WRITE !,"Transaction level at a=",$TLEVEL
&sql(SAVEPOINT b)
  WRITE !,"Set Savepoint b, SQLCODE=",SQLCODE
  WRITE !,"Transaction level at b=",$TLEVEL
&sql(ROLLBACK TO SAVEPOINT b)
  WRITE !,"Rollback to b, SQLCODE=",SQLCODE
  WRITE !,"Rollback transaction level=",$TLEVEL
&sql(SAVEPOINT c)
  WRITE !,"Set Savepoint c, SQLCODE=",SQLCODE
  WRITE !,"Transaction level at c=",$TLEVEL
&sql(SAVEPOINT d)
  WRITE !,"Set Savepoint d, SQLCODE=",SQLCODE
  WRITE !,"Transaction level at d=",$TLEVEL
&sql(COMMIT)
  WRITE !,"Commit transaction, SQLCODE=",SQLCODE
  WRITE !,"Transaction level=",$TLEVEL
```

Arguments

pointname

The name of an existing savepoint, specified as an identifier.

See Also

- SQL commands: COMMIT, SAVEPOINT, SET TRANSACTION, START TRANSACTION, $TLEVEL

- Transaction Processing

- SQLCODE error messages

- ObjectScript: TROLLBACK

- ObjectScript: Transaction Processing

SAVEPOINT (SQL)

Marks a point within a transaction.

Synopsis

```
SAVEPOINT pointname
```

Description

A **SAVEPOINT** statement marks a point within a transaction. Establishing a savepoint enables you to perform transaction roll back to the savepoint, undoing all work done and releasing all locks acquired during that period. In a long-running transaction, or a transaction with internal control structure, it is often desirable to be able to roll back part of the transaction without undoing all work submitted during the transaction.

The establishment of a savepoint increments the $TLEVEL transaction level counter. Rolling back to a savepoint decrements the $TLEVEL transaction level counter to its value immediately prior to the savepoint. You can establish up to 255 savepoints within a transaction. Exceeding this number of savepoints results in an SQLCODE -400 fatal error, a <TRANSACTION LEVEL> exception caught during SQL execution. The Terminal prompt displays the current transaction level as a `TLn:` prefix to the prompt, where *n* is an integer between 1 and 255 representing the current **$TLEVEL** count.

Each savepoint is associated with an savepoint name, a unique identifier. Savepoint names are not case-sensitive. A savepoint name can be a delimited identifier.

- If you specify a **SAVEPOINT** with no *pointname*, or with a *pointname* that is not a valid identifier or is an SQL Reserved Word, a runtime SQLCODE -301 error is issued.

- If you specify a **SAVEPOINT** with a *pointname* that begins with "SYS", a runtime SQLCODE -302 error is issued. These savepoint names are reserved.

Savepoint names are not case-sensitive; therefore `resetpt`, `ResetPt` and `"RESETPT"` are the same *pointname*. This duplication is detected during **ROLLBACK TO SAVEPOINT**, not during **SAVEPOINT**. When you specify a **SAVEPOINT** statement with a duplicate *pointname*, InterSystems IRIS increments the transaction level counter, just as if the *pointname* was unique. However, the most recent *pointname* overwrites all prior duplicate values in the table of savepoint names. Therefore, when you specify a **ROLLBACK TO SAVEPOINT** *pointname*, InterSystems IRIS rolls back to the most recently established **SAVEPOINT** with that *pointname*, and decrements the transaction level counter appropriately. However, if you again specify a **ROLLBACK TO SAVEPOINT** *pointname* with the same name, an SQLCODE -375 error is generated, with the %msg: `Cannot ROLLBACK to unestablished savepoint 'name'`, the full transaction is rolled back and the **$TLEVEL** count reverts to 0.

Using Savepoints

The **SAVEPOINT** statement is supported for Embedded SQL, Dynamic SQL, ODBC, and JDBC. In JDBC, `connection.setSavepoint(pointname)` sets a savepoint, and `connection.rollback(pointname)` rolls back to the named savepoint.

If savepoints have been established:

- A **ROLLBACK TO SAVEPOINT** *pointname* rolls back work done since the specified savepoint, deletes that savepoint and all intermediate savepoints, and decrements the **$TLEVEL** transaction level counter by the number of savepoints deleted. If *pointname* does not exist, or has already been rolled back, this command rolls back the entire transaction, resets **$TLEVEL** to 0, and releases all locks.

- A **ROLLBACK** rolls back all work done during the current transaction, rolling back the work done since **START TRANSACTION**. It resets the **$TLEVEL** transaction level counter to zero and releases all locks. Note that a generic **ROLLBACK** ignores savepoints.

- A **COMMIT** commits all work done during the current transaction. It resets the **$TLEVEL** transaction level counter to zero and releases all locks. Note that a **COMMIT** ignores savepoints.

Issuing a second **START TRANSACTION** within a transaction has no effect on savepoints or the **$TLEVEL** transaction level counter.

An SQLCODE -400 error is issued if a transaction operation fails to complete successfully.

Arguments

pointname

The name of the savepoint, specified as an identifier.

Examples

The following embedded SQL example creates a transaction with two savepoints:

ObjectScript

```
NEW SQLCODE,%ROWCOUNT,%ROWID
&sql(START TRANSACTION)
&sql(DELETE FROM Sample.Person WHERE Name=NULL)
IF SQLCODE=100 { WRITE !,"No null name records to delete" }
ELSEIF SQLCODE'=0 {&sql(ROLLBACK)}
ELSE {WRITE !,%ROWCOUNT," null name records deleted"}
  &sql(SAVEPOINT svpt_age1)
  &sql(DELETE FROM Sample.Person WHERE Age=NULL)
  IF SQLCODE=100 { WRITE !,"No null age records to delete" }
  ELSEIF SQLCODE'=0 {&sql(ROLLBACK TO SAVEPOINT svpt_age1)}
  ELSE {WRITE !,%ROWCOUNT," null age records deleted"}
    &sql(SAVEPOINT svpt_age2)
    &sql(DELETE FROM Sample.Person WHERE Age>65)
    IF SQLCODE=0 { &sql(COMMIT)}
    ELSEIF SQLCODE=100 { &sql(COMMIT)}
    ELSE {
     &sql(ROLLBACK TO SAVEPOINT svpt_age2)
     WRITE !,"retirement age deletes failed"
    }
  &sql(COMMIT)
&sql(COMMIT)
```

ObjectScript and SQL Transactions

ObjectScript transaction processing, using TSTART and TCOMMIT, differs from, and is incompatible with, SQL transaction processing using the SQL statements **START TRANSACTION**, **SAVEPOINT**, and **COMMIT**. Both ObjectScript and InterSystems SQL provides limited support for nested transactions. ObjectScript transaction processing does not interact with SQL lock control variables; of particular concern is the SQL lock escalation variable. An application should not attempt to mix the two types of transaction processing.

If a transaction involves SQL update statements, then the transaction should be started by the SQL **START TRANSACTION** statement and committed with the SQL **COMMIT** statement. Methods that use **TSTART**/**TCOMMIT** nesting can be included in the transaction, as long as they don't initiate the transaction. Methods and stored procedures should not normally use SQL transaction control statements, unless, by design, they are the main controller of the transaction.

See Also

- SQL commands: COMMIT ROLLBACK SET TRANSACTION START TRANSACTION $TLEVEL

- Transaction Processing

- SQLCODE error messages

- ObjectScript command: TCOMMIT

SELECT (SQL)

Retrieves rows from one or more tables within a database.

Synopsis

Basic Selection

```
SELECT * FROM table
SELECT selectItem FROM table
SELECT selectItem, selectItem2, ... FROM table
SELECT ... FROM table, table2, ...
```

Predicate Conditions

```
SELECT ... FROM ... WHERE condition
SELECT ... FROM ... [WHERE condition] GROUP BY column
SELECT ... FROM ... [WHERE condition][GROUP BY column]]
     HAVING condition
SELECT ... FROM ... [WHERE condition][GROUP BY column]
     [HAVING condition] ORDER BY itemOrder [ASC | DESC]
```

Aliases

```
SELECT selectItem AS columnAlias FROM ...
SELECT selectItem AS columnAlias, selectItem2 AS columnAlias2, ... FROM ...
SELECT ... FROM table AS tableAlias ...
SELECT ... FROM table AS tableAlias, table2 AS tableAlias2, ...
```

Selection Criteria

```
SELECT DISTINCT ... FROM ...
SELECT DISTINCT BY (distinctItem) ... FROM ...
SELECT DISTINCT BY (distinctItem, distinctItem2, ...) ... FROM ...
SELECT TOP numRows ... FROM ...
SELECT DISTINCT TOP ... FROM ...
SELECT TOP ALL ... FROM ...
SELECT ALL ... FROM ...
```

Embedded SQL Host Variables

```
SELECT selectItem INTO :var FROM ...
SELECT selectItem, selectItem2, ... INTO :var, :var2, ... FROM ...
SELECT * INTO :var() FROM ...
```

Keyword Options

```
SELECT %keyword ... FROM ...
```

Subqueries and Cached Queries

```
(SELECT ... FROM ...)
```

Sample Selection

```
SELECT ... FROM table WHERE %ID %FIND %SQL.SAMPLE( tablename, percent, seed )
```

Selection From a Time Series Machine Learning Model

```
SELECT WITH PREDICTIONS( model-name ) ... FROM ...
```

Description

The **SELECT** statement performs a query that retrieves data from an InterSystems IRIS® database. In its simplest form, it retrieves data from one or more columns of a single table. The **SELECT** *selectItem* clause specifies the columns to select. The **FROM** *table* clause specifies the table to select from, and the optional **WHERE** clause supplies one or more *condition* elements that determine which rows to return column values for.

In more complex queries, a **SELECT** statement can retrieve column data, aggregate data, computed column data, and data from multiple tables using joins. It can also retrieve data using views.

You can use a **SELECT** statement in these contexts:

- An independent InterSystems SQL query.

- A subquery that supplies values to an enclosing **SELECT** statement.

- A subset of a UNION. The UNION statement allows you to combine two or more **SELECT** statements into a single query.

- Part of a CREATE VIEW defining the data available to the view.

- Part of an INSERT with a SELECT statement. An **INSERT** statement can use a **SELECT** to insert data values for multiple rows into a table, selecting the data from another table. For more details, see Multi-Row Inserts.

- An independent query prepared as a Dynamic SQL query, Embedded SQL query, or Class Query. You can also use Dynamic SQL to return metadata about a **SELECT** query, such as the number of columns specified in the query, the name (or alias) of a column specified in the query, and the data type of a column specified in the query.

- Part of a DECLARE CURSOR used with Embedded SQL.

SELECT returns the results of a query in a *result set*. The command also sets a status variable, SQLCODE, indicating success or failure of the query. For more details, see SELECT Status and Return Values.

SELECT clauses must be specified in the order shown in the syntaxes. Specifying **SELECT** clauses in the incorrect order generates an SQLCODE -25 error. The **SELECT** syntax order is *not* the same as the **SELECT** clauses semantic processing order. For further details, refer to SELECT Clause Order of Execution.

Basic Selection

- **SELECT * FROM** *table* selects all items from a table. Typically, these items are the columns in the table.

  This query selects all columns from the Sample.Person table.

  SQL

  ```
  SELECT * FROM Sample.Person
  ```

  For more details on this format, see All Column Selections (Asterisk Syntax).

- **SELECT** *selectItem* **FROM** *table* selects a single item from a table.

  This query selects the Name column from the Sample.Person table.

  SQL

  ```
  SELECT Name FROM Sample.Person
  ```

- **SELECT** *selectItem*, *selectItem2*, ... **FROM** *table* selects multiple items from a table using a comma-separated list of *selectItem* values.

  This query selects the Name and Age columns from the Sample.Person table.

SQL

```
SELECT Name,Age FROM Sample.Person
```

- **SELECT ... FROM** *table*, *table2*, **...** selects the items from multiple tables using a comma-separated list of *table* names. If you try to select column names that exist in multiple tables without using an alias to specify which table to select the field from, the system raises a SQLCODE -27.

  This query selects the `SSN` and `Mission` columns from both the `Sample.Person` and `Sample.Company` table. For each `SSN`, it returns all `Missions`:

  SQL

  ```
  SELECT SSN,Mission FROM Sample.Person,Sample.Company
  ```

  To associate tables in a **SELECT** statement and select data from the intersection of the tables, use JOIN expressions.

Note: For statements that do not reference table data, such as ones that return data from functions, the **FROM** clause is optional. For more details on this clause, see FROM.

Predicate Conditions

- **SELECT ... FROM ... WHERE** *condition* returns the rows of the table for which *condition*, a set of predicates linked by logical operators, is true. For example, this statement selects only people who are over 40 and live in Massachusetts:

  SQL

  ```
  SELECT Name,Age,Home_State FROM Sample.Person WHERE Age > 40 AND Home_State = 'MA'
  ```

  The *condition* argument also limits the values supplied to aggregate functions to the values from those rows. A **WHERE** clause predicate does not directly accept aggregate functions. These values must be passed to the **WHERE** clause from other clauses. For more details, see WHERE.

  Example: Select Subsets of Data Using Predicate Conditions

- **SELECT ... FROM ... [WHERE** *condition***] GROUP BY** *column* organizes query result sets into groups, returning one row for each distinct value retrieved from the specified table columns.

  The *column* argument can be a comma-separated list of column names or a scalar expression that evaluates to a column name. The **GROUP BY** clause is often used in conjunction with aggregate functions.

  This query returns one row for each distinct state found in the `Home_State` column, with the computed `COUNT(Home_State)` selection returning the count of each state.

  SQL

  ```
  SELECT Home_State, COUNT(Home_State)
  FROM Sample.Person
  GROUP BY Home_State
  ```

  For more details on this clause, see GROUP BY .

  Example: Select Subsets of Data Using Predicate Conditions

- **SELECT ... FROM ... [WHERE** *condition***][GROUP BY** *column***] HAVING** *condition* returns the table rows for which **HAVING** *condition* is true. Unlike the **WHERE** clause, the **HAVING** clause operates on groups and is often used in combination with the **GROUP BY** clause.

  HAVING *condition* determines which rows are returned, but by default does not limit the values supplied to aggregate functions to the values from those rows. To override this default, use the `%AFTERHAVING` keyword.

Unlike **WHERE** clauses, **HAVING** clauses can specify aggregate functions.

This query returns rows for which Age is greater than average age of all people in the database.

SQL

```
SELECT Name,AVG(Age %AFTERHAVING)
FROM Sample.Person
HAVING (Age > AVG(Age))
```

For more details on this clause, see HAVING.

Example: Select Subsets of Data Using Predicate Conditions

- **SELECT ... FROM ... [WHERE** *condition***][GROUP BY** *column***][HAVING** *condition***] ORDER BY itemOrder [ASC | DESC]** specifies the order in which to display the returned rows. Specify *itemOrder* as a selection item specified in the **SELECT** *selectItem* clause or as a comma-separated list of such items.

  Each item can have an optional **ASC** (ascending order) or **DESC** (descending order) keyword specifying the order of the returned values for that item. The default is ascending order.

  This query returns the selected columns for all rows in the database in ascending order by age:

  SQL

  ```
  SELECT Home_State, Name, Age
  FROM Sample.Person
  ORDER BY Age
  ```

  The **SELECT** command applies the **ORDER BY** clause to the results of a query. This clause is frequently paired with a **TOP** clause. If you use an **ORDER BY** clause in a subquery or **CREATE VIEW** query, then the **TOP** clause is required.

  An **ORDER BY** clause can contain a window function, as described in Overview of Window Functions.

  For more details on this clause, see ORDER BY.

  Example: Select Subsets of Data Using Predicate Conditions

Aliases

- **SELECT** *selectItem* **AS** *columnAlias* **FROM ...** sets an alias for the name of a column or other *selectItem* value. The *columnAlias* value is displayed as the column head of the result set. Use aliases to make the returned data easier to understand. If you do not specify an alias, then the result set uses the column name specified in the select item.

  This query returns the Home_State results in a column titled US_State_Abbrev:

  SQL

  ```
  Select Home_State AS US_State_Abbrev FROM Sample.Person
  ```

  Example: Change Case of Columns in Result Set

- **SELECT** *selectItem* **AS** *columnAlias*, *selectItem2* **AS** *columnAlias2*, **... FROM ...** sets aliases for multiple select items.

  This query returns the results of Name and Home_State in PersonName and State columns, respectively.

  SQL

  ```
  Select Name AS PersonName,Home_State AS State FROM Sample.Person
  ```

  Example: Distinguish Between Column Names in Multi-Table Queries

- **SELECT ... FROM** *table* **AS** *tableAlias* **...** sets an alias for the table name. You can use table aliases as the table prefixes in *selectItem* values.

- **SELECT ... FROM** *table* **AS** *tableAlias*, *table2* **AS** *tableAlias2*, **...** sets an alias for multiple tables. You can use table aliases to indicate which table the selected column belong to. For example:

  SQL

  ```
  SELECT P.Name, E.Name FROM Sample.Person AS P, Sample.Employee AS E
  ```

  Example: Distinguish Between Column Names in Multi-Table Queries

Selection Criteria

DISTINCT Clause

- **SELECT DISTINCT ... FROM ...** returns only one row per unique combination of *selectItem* values. Use this clause to exclude redundant column values from the result set. You can specify one or more select items.

  This query returns one row per unique combination of `Home_State` and `Age` values:

  SQL

  ```
  SELECT DISTINCT Home_State,Age FROM Sample.Person
  ```

- **SELECT DISTINCT BY (***distinctItem***) ... FROM ...** returns one row per unique value of *distinctItem*, which must be an item from *selectItem*. Enclose *distinctItem* in parentheses. Use **DISTINCT BY** to return distinct values based on an item other than the one specified in *selectItem*. The query `SELECT DISTINCT BY (item) item FROM Sample.Person` is equivalent to `SELECT DISTINCT item FROM Sample.Person`.

  This query returns a row with `Name` and `Age` values for each unique `Age` value:

  SQL

  ```
  SELECT DISTINCT BY (Age) Name,Age FROM Sample.Person
  ```

- **SELECT DISTINCT BY (***distinctItem***,** ***distinctItem2***, ...) ... FROM ...** returns one row for each unique combination of *distinctItem* values using a comma-separated list.

  This query returns a row with `Name` and `Age` values for each unique combination of `Home_State` and `Age` values:

  SQL

  ```
  SELECT DISTINCT BY (Home_State,Age) Name,Age FROM Sample.Person
  ```

TOP Clause

- **SELECT TOP** *numRows* **... FROM ...** returns the specified number of rows, *numRows*, that appear at the "top" of the returned *table*. The default "top" rows can be unpredictable. For more control over the top rows returned, include a DISTINCT clause to return only unique values and an ORDER BY clause to order the values based on specific rows. The **SELECT** command applies these clauses before selecting the **TOP** rows.

  This query returns the top 10 `Name` values in alphabetical order.

  SQL

  ```
  SELECT TOP 10 Name FROM Sample.Person ORDER BY Name ASC
  ```

  For more details, see TOP.

- **SELECT DISTINCT TOP ... FROM ...** returns the unique "top" number of rows.

- **SELECT TOP ALL ... FROM ...** selects all rows. This syntax is only meaningful in a subquery or in a **CREATE VIEW** statement. It is used to support the use of an **ORDER BY** clause in these situations, fulfilling the requirement that an **ORDER BY** clause must be paired with a **TOP** clause in a subquery or a query used in a **CREATE VIEW** statement. The **TOP ALL** operation does not restrict the number of rows returned.

ALL Clause

- **SELECT ALL ... FROM ...** returns all rows that meet the **SELECT** criteria. This is the default for InterSystems SQL. The **ALL** keyword performs no operation and is provided only for SQL compatibility.

Embedded SQL Host Variables

- **SELECT** *selectItem* **INTO :***var* **FROM ...** selects a single column from a table and saves it into host variable *var*. You can supply host variables for Embedded SQL queries only.

- **SELECT** *selectItem*, *selectItem2*, ... **INTO :***var*, **:***var2*, ... **FROM ...** selects multiple columns and saves them to corresponding host variables. The number of columns must match the number of host variables. You can also use host variables in predicate clauses.

  This class snippet declares a cursor that selects two fields and stores them in host variables for later fetches. The cursor sorts and filters the fetched results by storing the class input arguments as separate host variables.

  ### Class Member

  ```
  ClassMethod AgeThreshold(ageThreshold As %Integer, orderBy As %String = "") As %Status
  {
    write "People who are age " _ageThreshold_ " and up:"

    &sql(declare CC cursor for
    SELECT Name,Age
    INTO :name,:age
    FROM Demo.Person
    WHERE (Age >= :ageThreshold)
    ORDER BY :orderBy)

    // ...
  }
  ```

  Example: Select Data from Within ObjectScript Programs Using Embedded SQL and Dynamic SQL

- **SELECT * INTO :***var***() FROM ...** selects all columns from a table and saves them to the subscripted variable *var*. Columns specified as private in the class definition are not included. To access the host variables, use the syntax *var*(*colIndex*), where *colIndex* is the column order index as determined by the SqlColumnNumber of the column. For more details, see Host Variable Subscripted by Column Number.

Keyword Options

- **SELECT** *%keyword* **... FROM ...** sets one or more *%keyword* options, separated by spaces. Valid options are `%NOFPLAN`, `%NOLOCK`, `%NORUNTIME`, `%PROFILE`, and `%PROFILE_ALL`. To use a *%keyword* argument, you must have the corresponding administration privilege for the current namespace. For more details, see GRANT.

Subqueries and Cached Queries

- **(SELECT ... FROM ...)** generates a separate cached query for each set of parentheses added. Queries in parentheses are also a requirement for specifying subqueries.

  In **SELECT** statements, you can specify a subquery in the *selectItem* list, the FROM clause, or in the **WHERE** clause with an EXISTS or IN predicate. You can also specify a subquery in an UPDATE or DELETE statement. A subquery must be enclosed in parentheses.

One or more sets of parentheses are optional for independent **SELECT** queries, **UNION** subset **SELECT** queries, **CREATE VIEW SELECT** queries, and **DECLARE CURSOR SELECT** queries. Enclosing a **SELECT** query in parentheses causes it to follow the syntax rules for a subquery. Specifically, you must pair an **ORDER BY** clause with a **TOP** clause.

Parentheses are not permitted for an **INSERT ... SELECT** statement.

Sample Selection

- **SELECT ... FROM** *table* **WHERE %ID %FIND %SQL.SAMPLE(** *tablename*, *percent*, *seed* **)** selects a random sample from the table. The *percent* argument specifies what percentage of each data block to sample; since the number of rows stored in a data block is variable, sampling 10% of the table does not necessarily return 10% of all the rows in the table. *table* and *tablename* must be the qualified name of the table you wish to sample from, but *table* must be an identifier and *tablename* must be a string.

  Additionally, *table* cannot refer to a view, must have a bitmap extent index, and must be block-sampling enabled. The RowID field must be declared or implied to be a positive integer, as is the default behavior.

  The *seed* argument is optional. If it is not provided, the default is the empty string.

Selection From a Time Series Machine Learning Model

- **SELECT WITH PREDICTIONS(** *model-name* **) ... FROM ...** selects the specified columns from a time series model, appending or pre-pending any predicted rows to the results, depending on the prediction window you specified when creating the model. You can use a WHERE clause to limit the results to the predicted rows by filtering on column that the time series was created from.

  For more information about IntegratedML, see "Introduction to IntegratedML."

Arguments

selectItem

selectItem is a mandatory argument for all **SELECT** statements that specifies an item, or comma-separated list of items to select from tables. You can specify each *selectItem* as one of the following:

- The column name of a table specified in the **FROM** clause. See Table Column Selections.

- A subquery that references table columns or an entire table. See Subquery Selections.

- The column name of a table that is not specified in the **FROM** clause, using implicit joins (also known as *arrow syntax*). See Implicit Join Selections.

- All columns in a table, using the asterisk (*) syntax. See All Column Selections.

- Properties in an embedded serial object, using the underscore (_) syntax. See Embedded Serial Object Selections.

You can also specify *selectItem* as a function that modifies data selected from table columns or computes new data from the selection. You can specify these items:

- A SQL function that aggregates selected column data and returns a single value. See Aggregate Function Selections.

- A window function that calculates aggregates, ranking, and other functions for each row, based on a "window frame" specific to that row. See Window Function Selections.

- An SQL function, ObjectScript class method call, or ObjectScript function call that operates on selected table data. See Function and Method Call Selections.

Finally, you can use *selectItem* to generate columns with the same value for all returned rows. You can also insert the same text or other data into each row of a selected column. These selections do not operate on the table data. See Non-Table Data Selections.

Table Column Selections

Commonly, a *selectItem* refers to a column in the *table* specified in the **FROM** clause. To specify multiple columns, use a comma-separated list. For example:

SQL

```
SELECT Name,Age FROM Sample.Person
```

Column names are not case-sensitive. However, the label associated with the column in the result set does not use the letter case specified in *selectItem*. Instead, it uses the letter case of the corresponding SqlFieldName ObjectScript property, as specified in the table definition. For more details on letter case resolution, see Change Case of Columns in Result Set example.

To list all of the column names defined for a specified table, see Column Names and Numbers.

To display the RowID (record ID), you can use the %ID pseudo-field variable alias, which displays the RowID regardless of what name it is assigned. By default, the name of the RowID is ID, but if the table already includes a column named ID, then InterSystems IRIS might rename it. By default, RowID is a hidden column.

Running a **SELECT** query on a stream column returns the OREF (object reference) of the opened stream object. For example:

SQL

```
SELECT Name,Picture FROM Sample.Employee WHERE Picture IS NOT NULL
```

When the **FROM** clause specifies more than one table or view, you must include the table name as part of the *selectItem*, separating the table name and column name with a period. For example:

SQL

```
SELECT Sample.Person.Name,Sample.Employee.Company
    FROM Sample.Person, Sample.Employee
```

If you specified a table alias, specify that alias in the *selectItem* instead. For example:

SQL

```
SELECT p.Name, e.Company
    FROM Sample.Person AS p, Sample.Employee AS e
```

If a table name already has an assigned alias, specifying the full table name as part of a *selectItem* results in an SQLCODE -23 error.

Subquery Selections

Specifying *selectItem* as a subquery returns a single column from a specified table. This column can contain the values of a single table column or of multiple table columns returned as a single column. You can return multiple columns in a single column by using either concatenation (**SELECT Home_City||Home_State**) or by specifying a container column (**SELECT Home**). A subquery can use implicit joins (arrow syntax). A subquery cannot use asterisk syntax, even when the table cited in the subquery has only one data column.

You can use subqueries to specify an aggregate function that is not subject to the **GROUP BY** clause. In the following example, the **GROUP BY** clause groups ages by decades (for example, 25 through 34). The **AVG(Age)** *selectItem* gives

the average age of each group, as defined by the **GROUP BY** clause. To get the average age of all records across all groups, it uses a subquery:

SQL

```
SELECT Age AS Decade,
       COUNT(Age) AS PeopleInDecade,
       AVG(Age) AS AvgAgeForDecade,
       (SELECT AVG(Age) FROM Sample.Person) AS AvgAgeAllDecades
FROM Sample.Person
GROUP BY ROUND(Age,-1)
ORDER BY Age
```

Implicit Join Selections (Arrow Syntax)

To access a column from a table other than the **FROM** clause table, you can specify *selectItem* as an *implicit join* by using the arrow syntax (->). In the following example, the Sample.Employee table contains a Company column, which in turn contains the RowID for the corresponding company name in the Sample.Company table. The arrow syntax retrieves the company name from that table:

SQL

```
SELECT Name,Company->Name AS CompanyName
       FROM Sample.Employee
```

In this case, you must have **SELECT** privileges for the referenced table: either table-level **SELECT** privileges or column-level **SELECT** privileges for both the referenced column and the RowID column of the referenced table. For more details on arrow syntax, see Implicit Joins (Arrow Syntax).

All Column Selections (Asterisk Syntax)

To select all the columns in a table, use the asterisk syntax (*). For example:

SQL

```
SELECT TOP 5 * FROM Sample.Person
```

Items are returned in column number order. Asterisk syntax selections include embedded serial object properties, including properties from a serial object nested within a serial object. A column referencing a serial object is not selected. For example, the Home_City property from an embedded serial object is selected, but the Home referencing column used to access the Sample.Address embedded serial class, which contains the City property, is not selected.

Asterisk syntax does not select hidden columns. By default, the RowID is hidden (not displayed by SELECT *). However, if the table was defined with %PUBLICROWID, then SELECT * returns the RowID column and all non-hidden columns. By default, the name of this column is ID, but if a user-defined column named ID already exists, InterSystems IRIS might rename it.

A query with an asterisk and more than one *table* selects all the columns in all the joined tables. For example, this query selects all columns for the top 5 rows of both Sample.Company and Sample.Employee.

SQL

```
SELECT TOP 5 * FROM Sample.Company,Sample.Employee
```

The asterisk syntax can be qualified or unqualified. If the *selectItem* is qualified by prefixing a table name (or table name alias) and period (.) before the asterisk, the *selectItem* selects all the columns in the specified table. You can combine the qualified asterisk syntax with other select items for other tables. In this example, *selectItem* consists of an unqualified asterisk syntax that selects all columns from the table. Note that you can also specify duplicate column names (in this case Name) and non-column *selectItem* elements (in this case {fn NOW}):

SQL

```
SELECT TOP 5 {fn NOW} AS QueryDate,
             Name AS Client,
             *
FROM Sample.Person
```

In this example, *selectItem* consists of the qualified asterisk syntax that selects all columns from one table and a list of column names from another table.

SQL

```
SELECT TOP 5 E.Name AS EmpName,
             C.*,
             E.Home_State AS EmpState
FROM Sample.Employee AS E, Sample.Company AS C
```

Note: SELECT * is a fully supported part of InterSystems SQL that can be extremely convenient during application development and debugging. However, in production applications, the preferred programming practice is to explicitly list the selected columns. Explicitly listing columns makes your application clearer and easier to understand, easier to maintain, and easier to search for columns by name.

Embedded Serial Object Selections (Underscore Syntax)

To select an embedded serial object property (embedded serial class data), specify *selectItem* using underscore syntax. Underscore syntax consists of the name of the object property, an underscore, and the property within the embedded object: for example, `Home_City` and `Home_State`. (In other contexts, index tables for example, these are represented using dot syntax: `Home.City`.)

Consider this example:

SQL

```
SELECT Home_City,Home_Phone_AreaCode FROM Sample.Person
```

For the column name `Home_City`, the `Sample.Person` table contains a referencing column `Home`. This column references an embedded serial object that defines the property `City`. For the column name `Home_Phone_AreaCode`, the table contains a referencing column `Home` that references an embedded serial object property `Phone` that references a nested embedded serial object that defines the property `AreaCode`. If you select a referencing column such as `Home` or `Home_Phone`, you receive the values of all of properties in the serial object in %List data type format.

You can use **SELECT** to directly query a referencing column (such as `Home`), rather than using the underscore syntax. Because the data returned is in list format, you can use a **$LISTTOSTRING** or **$LISTGET** function to display the data. For example:

SQL

```
SELECT $LISTTOSTRING(Home,'^') AS HomeAddress FROM Sample.Person
```

Aggregate Function Selections

A *selectItem* can contain one or more SQL aggregate functions. An aggregate function always returns a single value. This table shows the type of aggregate functions that you can specify.

Aggregate Function Type	Example
A single column name, which computes the aggregate for all non-null values of the rows selected by the query.	`SELECT AVG(Age) FROM Sample.Person`
A scalar expression that computes an aggregate.	`SELECT SUM(Age) / COUNT(*) FROM Sample.Person`
Asterisk syntax (*), used with the **COUNT** function to compute the number of rows in the table.	`SELECT COUNT(*) FROM Sample.Person`
A **SELECT ... DISTINCT** function, which computes the aggregate by eliminating redundant values.	`SELECT COUNT(DISTINCT Home_State) FROM Sample.Person`
A combination of column names and aggregate functions in a single **SELECT** statement (allowed in InterSystems SQL but not ANSI SQL).	`SELECT Name, COUNT(DISTINCT Home_State) FROM Sample.Person`
An aggregate function using `%FOREACH`, which computes the aggregate for each distinct value of a column or columns.	`SELECT DISTINCT Home_State, AVG(Age %FOREACH(Home_State)) FROM Sample.Person`
An aggregate function using `%AFTERHAVING`. This causes the aggregate to be computed on a sub-population specified with the **HAVING** clause. In the example shown, the query returns records where `Age` is greater than average age of all people in the database.	`SELECT Name,AVG(Age %AFTERHAVING) FROM Sample.Person HAVING (Age > AVG(Age))`

Window Function Selections

Specify *selectItem* as a window function to calculate aggregates, rankings, and other functions for each row, based on a "window frame" specific to that row. The following syntax is supported:

```
windowFunction() OVER (
    PARTITION BY partColumn
    ORDER BY orderColumn)
```

- *windowFunction*: The following window functions are supported: `AVG()`, `ROW_NUMBER()`, `RANK()`, `PERCENT_RANK()`, `FIRST_VALUE(column)`, `LAST_VALUE(column)`, `NTH_VALUE(column, n)`, `LAG(column, offset)`, `LEAD(column, offset)`, `MAX(column)`, `MIN(column)`, and `SUM(column)`.

- **OVER**: The **OVER** keyword followed by parentheses is mandatory. Clauses within these parentheses are optional.

- **PARTITION BY** *partColumn*: An optional clause that partitions rows by the specified *partColumn*. The *partColumn* argument can be a single column or a comma-separated list of columns. If specified, **PARTITION BY** must be specified before **ORDER BY**.

- **ORDER BY** *orderColumn*: An optional clause that orders rows by the specified *orderColumn*. The *orderColumn* can be a single column or a comma-separated list of columns.

Columns specified in a window function can take a table alias prefix.

A window function can specify a column alias. By default the column is labeled `Window_n`.

For more details, see Overview of Window Functions.

Function and Method Call Selections

A *selectItem* can apply additional processing to the column values it selects by using function and method operations. You can specify these operation types:

- An arithmetic operation. For example, this selection generates a new column by subtracting the average age from the `Age` column.

   SQL

   ```
   SELECT Name, Age,Age-AVG(Age) FROM Sample.Person
   ```

   If a *selectItem* arithmetic operation includes division, and any column value produces a divisor of 0 or NULL, you cannot rely on order of testing to avoid division by zero. Instead, use a case statement to suppress the risk.

- An SQL function. For example, this query generates a column for the length of each value in the `Name` column:

   SQL

   ```
   SELECT Name,$LENGTH(Name) FROM Sample.Person
   ```

   This query converts the case of the `Name` column to uppercase and returns it in a new column.

   SQL

   ```
   SELECT Name,UCASE(Name) FROM Sample.Person
   ```

- An **XMLELEMENT**, **XMLFOREST**, or **XMLCONCAT** function, which place XML or HTML tags around the data values retrieved from specified column names. For more details, see XMLELEMENT.

- A collation function, which specifies the sorting and display of a *selectItem* column. You can supply the collation function without parentheses (`SELECT %SQLUPPER Name`) or with parentheses (`SELECT %SQLUPPER(Name)`). If the collation function specifies truncation, the parentheses are required (`SELECT %SQLUPPER(Name,10)`).

- A user-defined class method stored as a procedure. The class method can be an unqualified method name (for example, `RandLetter()`) or a qualified method name (for example, `Sample.RandLetter()`). In this class, the `Cube()` class method returns the cube of the input integer:

   Class Definition

   ```
   Class Sample.Person Extends %Persistent [DdlAllowed]
   {
   /// Find the Cube of a number
   ClassMethod Cube(val As %Integer) As %Integer [SqlProc]
      {
        RETURN val * val * val
      }
   }
   ```

   This query calls the `Cube` class method on the `Age` column to return the cubed age.

   SQL

   ```
   SELECT Age, Person_Cube(Age) FROM Sample.Person
   ```

   InterSystems IRIS coverts the method return value from `Logical` to `Display/ODBC` format. By default, inputs to the method are not converted from `Display/ODBC` to `Logical` format. However,you can configure input display-

to-logical conversion system-wide using the **$SYSTEM.SQL.Util.SetOption("SQLFunctionArgConversion")** method. To determine the current configuration of this option, use **$SYSTEM.SQL.Util.GetOption("SQLFunctionArgConversion")**.

If the specified method does not exist in the current namespace, the system generates an SQLCODE -359 error. If the specified method is ambiguous, meaning it could refer to more than one method, the system generates an SQLCODE -358 error. For more details on class method creation, see CREATE METHOD.

- A user-supplied ObjectScript function call (extrinsic function) operating on a database column. For example:

  SQL

  ```
  SELECT $$REFORMAT(Name)FROM MyTable
  ```

  To call such functions in an SQL statement, you must configure the `Allow extrinsic functions in SQL statements` option system-wide. For more details, see Functions: Intrinsic and Extrinsic. By default, extrinsic functions are disabled and attempting to call user-supplied functions generates an SQLCODE -372 error.

  Trying to use a user-supplied function to call a % routine generates an SQLCODE -373 error.

Non-Table Data Selections

The *selectItem* argument can return the same value for all records without referencing the table that is in the **FROM** clause. When no *selectItem* elements reference table data, the **FROM** clause is optional. If you include the **FROM** clause, the specified table must exist. For more details, see FROM.

Common uses for this format selection are as follows:

- Arithmetic operations.

  SQL

  ```
  SELECT Name, Age, 9 – 6 FROM Sample.Person
  ```

- A string literal or a function operating on a string literal.

  SQL

  ```
  SELECT UCASE('fred') FROM Sample.Person
  ```

- A string literal added to produce a more readable output.

  SQL

  ```
  SELECT TOP 10 Name,'was born on',%EXTERNAL(DOB)
  FROM Sample.Person
  ```

  How you specify the numeric literal determines its data type. For example, the string `'123'` is of data type `VARCHAR`, and the numeric value `123` is of data type `INTEGER` or `NUMERIC`.

- A %TABLENAME, or %CLASSNAME pseudo-field variable keyword. %TABLENAME returns the current table name. %CLASSNAME returns the name of the class corresponding to the current table. If the query references multiple tables, you can prefix the keyword with a table alias. For example, `t1.%TABLENAME`.

- One of the following ObjectScript special variables (or their abbreviations): $HOROLOG, $JOB, $NAMESPACE, $TLEVEL, $USERNAME, $ZHOROLOG, $ZJOB, $ZNSPACE, $ZPI, $ZTIMESTAMP, $ZTIMEZONE, $ZVERSION.

table

One or more tables, views, table-valued functions, or subqueries from which data is being retrieved. You can specify any combination of these *table* types as a comma-separated list or with the JOIN syntax.

- If you specify a single *table* name, the specified data is retrieved from that table or view.

- If you specify multiple *table* names, InterSystems SQL performs a join operation on the tables, merging their data into a results table from which the specified data is retrieved.

A valid table reference is required for every **FROM** clause, even if the **SELECT** makes no reference to that table.

- To determine whether a table or view exists in the current namespace, use the **$SYSTEM.SQL.Schema.TableExists("schema.tname")** or **$SYSTEM.SQL.Schema.ViewExists("schema.vname")** method.

- To determine if you have SELECT privileges for a table or view, use the **$SYSTEM.SQL.Security.CheckPrivilege()** method

table can be either qualified (schema.tablename) or unqualified (tablename). An unqualified *table* is supplied either the default schema name or a schema name from the schema search path.

You can optionally assign an alias, *tableAlias*, to each *table*.

You can optionally specify one or more *optimize-option* keywords to optimize query execution. The available options are: `%ALLINDEX`, `%FIRSTTABLE`, `%FULL`, `%INORDER`, `%IGNOREINDEX`, `%NOFLATTEN`, `%NOMERGE`, `%NOREDUCE`, `%NOSVSO`, `%NOTOPOPT`, `%NOUNIONOROPT`, `%PARALLEL`, and `%STARTTABLE`. For more details on these options, see FROM.

condition

Logical tests (predicates) used in WHERE and HAVING clauses to specify the rows of data to retrieve. In **SELECT** statements, **WHERE** *condition* and **HAVING** *condition* return the rows for which *condition* evaluates to true.

To combine logical predicate conditions, use **AND** and **OR** logical operators. To invert a condition, use the **NOT** unary logical operator.

This table shows sample predicate conditions.

Predicate	Description	Example
Equality Comparisons	Return rows using =, <, >, and other comparison operators.	`SELECT Name,Age FROM Sample.Person WHERE Age < 21`
BETWEEN	Return rows between certain values	`SELECT Name,Age FROM Sample.Person WHERE Age BETWEEN 18 AND 21`
IN and %INLIST	Return rows that match items in a list.	`SELECT Name, Home_State FROM Sample.Person WHERE Home_State IN ('ME','NH,'VT')`
Substring Comparisons	Return rows that match a substring	`SELECT Name FROM Sample.Person WHERE Name %STARTSWITH 'S'`
NULL	Return rows based on the detection of undefined values	`SELECT Name,Age FROM Sample.Person WHERE Age IS NOT NULL`
EXISTS	Return rows based on the existence of at least one row in a table. Often used with subqueries.	`SELECT Name FROM Sample.Person WHERE EXISTS (SELECT * FROM Employee WHERE Employee.Number = Person.Number)`
FOR SOME	Return rows based on a condition test of certain column values. Often used to test whether a value in one table appears in another table.	`SELECT Name,COUNT(Name) FROM Sample.Person WHERE FOR SOME (Sample.Employee)(Sample.Employee.Name=Sample.Person.Name)`
FOR SOME %ELEMENT	Return rows that match certain list element values.	`SELECT Name,FavoriteColors FROM Sample.Person WHERE FOR SOME %ELEMENT(FavoriteColors) (%VALUE='Red')`
LIKE, %MATCHES, and %PATTERN	Match row that fit a specific pattern.	`SELECT Name FROM Sample.Person WHERE Name LIKE '%Mac%'`

For more details on these logical predicates, see WHERE.

condition cannot contain aggregate functions. To specify a selection condition using a value returned by an aggregate function, use a HAVING clause.

In **WHERE** clauses, *condition* can specify an explicit join between two tables using the = (inner join) symbolic join operators. For more details, see JOIN.

A **WHERE** clause can specify an implicit join between the base table and a column from another table using the arrow syntax (–>) operator. For more details, see Implicit Joins.

column

A comma-separated list of columns specifying how to organize retrieved data. Valid *column* values include:

- A column name (GROUP BY City)

- An %ID (returns all rows)

- A scalar function specifying a column name (GROUP BY ROUND(Age,-1))

- A collation function specifying a column name (GROUP BY %EXACT(City))

For more details, see GROUP BY.

itemOrder

A selectItem or a comma-separated list of items that specify the order in which rows are displayed. Each item can have an optional **ASC** (ascending order) or **DESC** (descending order) keyword. The default is ascending order. The **ORDER BY** clause operates on the results of a query. An **ORDER BY** clause in a subquery, such as in a **UNION** statement, must be paired with a **TOP** clause. If no **ORDER BY** clause is specified, the order of the records returned is unpredictable. An **ORDER BY** clause can include window functions. For more details, see ORDER BY.

columnAlias

In **SELECT** queries, each column in selectItem can have an alias. The column alias is displayed as the column header in the result set. If you do not specify a column alias, the name of the select item is used as the column name in the result set. The **AS** keyword separates the *selectItem* from the *columnAlias*. This keyword is optional but recommended for readability. Therefore, these syntaxes are equivalent and valid:

```
SELECT Name AS PersonName, DOB AS BirthDate FROM Sample.Person
SELECT Name PersonName, DOB BirthDate FROM Sample.Person
```

InterSystems SQL displays column aliases with the specified letter case, but aliases are not case-sensitive when referenced in an **ORDER BY** clause. The *columnAlias* name must be a valid identifier, including a delimited identifier. Using a delimited identifier permits a column alias to contain spaces, other punctuation characters, or to be an SQL reserved name (for example, SELECT Name AS "Customer Name" or SELECT Home_State AS "From").

SQL does not perform uniqueness checking for column aliases. It is possible (though not desirable) for a column and a column alias to have the same name, or for two column aliases to be identical. Such non-unique column aliases can cause an SQLCODE -24 "Ambiguous sort column" error when referenced by an **ORDER BY** clause. Column aliases, like all SQL identifiers, are not case-sensitive.

Use of column aliases in other **SELECT** clauses is governed by query semantic processing order. You can reference a column by its column alias in an ORDER BY clause.

Referencing a column alias in these places is not allowed:

- Another *selectItem* in the select list

- **DISTINCT BY** clause

- **WHERE** clause

- **GROUP BY** clause

- **HAVING** clause

- **ON** or **USING** clause of a **JOIN** operation

You can, however, use a subquery to make a column alias available for use by these other **SELECT** clauses, as described in Querying the Database.

In addition to setting column aliases, you can also set aliases for aggregate functions, expressions, or other computed columns. Computed columns are automatically assigned a column name. If you do not provide an alias, InterSystems SQL supplies a unique column name, such as Expression_1 or Aggregate_3. The integer suffix refers to the *selectItem*

position as specified in the **SELECT** statement (that is, the *selectItem* column number). These values are *not* a count of columns of that type.

The following list shows the automatically assigned column names, where *n* is an integer. These names are listed in increasingly inclusive order. For example, adding a plus or minus sign to a number promotes it from a `HostVar` to an `Expression`; concatenating a `HostVar` and a `Literal` promotes it to an `Expression`; specifying a `Literal`, `HostVar`, `Aggregate`, or `Expression` in a subquery promotes it to a `SubQuery`:

- `Literal_n`: A pseudo-field variable such as %TABLENAME, or the NULL specifier. Note that %ID is not `Literal_n`; it is given the column name of the actual `RowID` column.

- `HostVar_n`: a host variable. This can be a literal, such as 'text', 123, or the empty string ("), an input variable (`:myvar`), or a ? input parameter replaced by a literal. Any expression evaluation on a literal, such as appending a sign to a number, string concatenation, or an arithmetic operation, makes it an `Expression_n`. A literal value supplied to a ? parameter is returned unchanged without expression evaluation. For example, supplying 5+7 returns the string `'5+7'` as `HostVar_n`.

- `Aggregate_n`: An aggregate function, such as `AVG(Age)` or `COUNT(*)`. A column is named `Aggregate_n` if the outermost operation is an aggregate function, even when this aggregate contains an expression. For example, `COUNT(Name)+COUNT(Spouse)` is `Expression_n`, but `MAX(COUNT(Name)+COUNT(Spouse))` is `Aggregate_n`, `-AVG(Age)` is `Expression_n`, but `AVG(-Age)` is `Aggregate_n`. In this example, the aggregate column created by the **AVG** function is given the column alias `AvgAge`. Its default name is `Aggregate_3` (an aggregate column in position 3 in the **SELECT** list).

  SQL

  ```
  SELECT Name, Age, AVG(Age) AS AvgAge FROM Sample.Person
  ```

- `Expression_n`: Any operation in the *selectItem* list on a literal, a column, or on an `Aggregate_n`, `HostVar_n`, `Literal_n`, or `Subquery_n` *selectItem* changes its column name to `Expression_n`. This includes unary operations on numbers (`-Age`), arithmetic operations (`Age+5`), concatenation (`'USA:'||Home_State`), data type **CAST** operations, SQL collation functions (`%SQLUPPER(Name)` or `%SQLUPPER Name`), SQL scalar functions (`$LENGTH(Name)`), user-defined class methods, **CASE** expressions, and special variables (such as `CURRENT_DATE` or `$ZPI`).

- `Window_n`: The result of a window function. You specify the column alias after the closing parenthesis of the **OVER** keyword.

- `Subquery_n`: The result of a subquery that specifies a single *selectItem*. The *selectItem* can be a column, aggregate function, expression, or literal. Specify the column alias after, not within, the subquery. For example:

  SQL

  ```
  SELECT Name AS PersonName,
         (SELECT Name FROM Sample.Employee) AS EmpName,
         Age AS YearsOld
  FROM Sample.Person
  ```

tableAlias

In a **SELECT** statement, you can specify an optional alias for a table or view name (*table*) as a valid identifier, including a delimited identifier. The **AS** keyword separates the *table* from the *tableAlias*. This keyword is optional but recommended for readability. Therefore, these syntaxes are equivalent and valid:

```
SELECT P.Name FROM Sample.Person AS P
SELECT P.Name FROM Sample.Person P
```

A *tableAlias* must be unique among table aliases within the query. A *tableAlias*, like all identifiers, is not case-sensitive. Specifying two *tableAlias* names that differ only in letter case results in an SQLCODE -20 "Name conflict" error.

The table alias is used as a prefix (with a period) to a column name to indicate the table to which the column belongs. For example:

SQL

```
SELECT P.Name, E.Name
FROM Sample.Person AS P, Sample.Employee AS E
```

When a query specifies multiple tables that have the same column name, you must use a table reference prefix. A table reference prefix can be a *tableAlias*, as shown in the previous example, or a fully qualified table name, as shown in this equivalent example:

SQL

```
SELECT Sample.Person.Name, Sample.Employee.Name
FROM Sample.Person, Sample.Employee
```

If you assign a *tableAlias* to a table name, then specifying a full table name as part of a *selectItem* results in an SQLCODE -23 error. Table aliases are required or optional depending on the query scenario.

Scenario	Table Alias
A query references only one table.	Optional
A query references multiple tables and the column names referenced are unique to each table.	Optional (but recommended)
A query references multiple tables and the column names referenced are the same in different tables.	Required
	Failing to specify a *tableAlias* (or fully qualified table name) prefix results in an SQLCODE -27 "Field %1 is ambiguous among the applicable tables" error.

You can also optionally use a *tableAlias* when specifying a subquery like this one:

SQL

```
SELECT Name,(SELECT Name FROM Sample.Vendor)
FROM Sample.Person
```

A *tableAlias* only uniquely identifies a column for query execution. To uniquely identify a column for query result set display, you must also use a column alias (*columnAlias*). This query combines the use of table aliases (`Per` and `Emp`) and column aliases (`PName` and `Ename`):

SQL

```
SELECT Per.Name AS PName, Emp.Name AS EName
FROM Sample.Person AS Per, Sample.Employee AS Emp
WHERE Per.Name %STARTSWITH 'G'
```

You can use the same name for a column, a column alias, and/or a table alias without a naming conflict.

Use the *tableAlias* prefix to distinguish which table is being referred to. For example:

SQL

```
SELECT P.%ID As PersonID,
       AVG(P.Age) AS AvgAge,
       Z.%TABLENAME||'=' AS Tablename,
       Z.*
FROM Sample.Person AS P, Sample.USZipCode AS Z
WHERE P.Home_City = Z.City
GROUP BY P.Home_City
ORDER BY Z.City
```

distinctItem

A comma-separated list of selectItem columns from which you want to exclude redundant rows in the result set. The *distinctItem* argument accepts any valid *selectItem* value. It does not accept the asterisk (*) keyword that selects all items. It also does not accept column name aliases.

Either type of DISTINCT clause can specify more than one item to test for uniqueness. Listing more than one item retrieves all rows that are distinct for the combination of both items. DISTINCT does consider NULL a unique value. For more details, see DISTINCT.

numRows

The number of rows to return, when used in conjunction with a **TOP** clause, as in **TOP** *numRows*. If the query does not contain an **ORDER BY** clause, the returned "top" rows is unpredictable. If the query contains an **ORDER BY** clause, the top rows are based on the specified order. If the query includes the **DISTINCT** keyword before **TOP**, then the query returns *numRows* unique values. Specify *numRows* as either a positive integer or a Dynamic SQL input parameter using the question mark (?) syntax that resolves to a positive integer. If no **TOP** keyword is specified, the default is to display all the rows that meet the **SELECT** criteria.

var

One or more host variables into which you place selectItem values. Specify multiple host variables as a comma-separated list or as a single-host variable array. For more details, see INTO.

Specifying an **INTO** clause in a **SELECT** query processed via ODBC, JDBC, or Dynamic SQL results in an SQLCODE -422 error.

%keyword

One or more *%keyword* arguments, separated by spaces. These keywords affect processing as follows:

- %NOFPLAN — The frozen plan (if any) is ignored for this operation; the operation generates a new query plan. The frozen plan is retained but not used. For more details, see Frozen Plans.

- %NOLOCK — InterSystems IRIS performs no locking on any of the tables. If you specify this keyword, the query retrieves data in READ UNCOMMITTED mode, regardless of current transaction's isolation mode. For more details, see Transaction Processing.

- %NORUNTIME — Runtime Plan Choice (RTPC) optimization is not used.

- %PROFILE or %PROFILE_ALL — Generate SQLStats collecting code. This is the same code that would be generated with PTools turned ON. The difference is that SQLStats collecting code is only generated for this specific statement. All other SQL statements within the routine or class being compiled generate code as if PTools is turned OFF. This enables you to profile and inspect specific problem SQL statements within an application without collecting irrelevant statistics for SQL statements that are not being investigated. For further details, see SQL Performance Analysis Toolkit.

  %PROFILE collects SQLStats for the main query module. %PROFILE_ALL collects SQLStats for the main query module and all its subquery modules.

Examples

Select Subsets of Data Using Predicate Conditions

Select subsets of data from a table using different combinations of predicate conditions. The clauses shown in these examples must be specified in the correct order. In all four examples, you select three columns (Name, Home_State, and Age) from the Sample.Person table and compute two other columns (AvgAge and AvgMiddleAge).

HAVING and ORDER BY

This query computes the AvgAge column on all records in Sample.Person. The **HAVING** clause governs the AvgMiddleAge computed column, calculating the average age of people over 40 from all records in Sample.Person. Thus, every row has the same value for AvgAge and AvgMiddleAge. The **ORDER BY** clause sequences the display of the rows alphabetically by the Home_State column value.

SQL

```
SELECT Name,Home_State,Age,AVG(Age) AS AvgAge,
 AVG(Age %AFTERHAVING) AS AvgMiddleAge
 FROM Sample.Person
 HAVING Age > 40
 ORDER BY Home_State
```

WHERE, HAVING, and ORDER BY

In this query, the **WHERE** clause limits the selection to the seven specified northeastern states. The query computes the AvgAge column on the records from those states. The **HAVING** clause governs the AvgMiddleAge computed column, calculating the average age of those over 40 from the records from the specified Home_State column. Thus, every row has the same value for AvgAge and AvgMiddleAge. The **ORDER BY** clause sequences the display of the rows alphabetically by the Home_State column value.

SQL

```
SELECT Name,Home_State,Age,AVG(Age) AS AvgAge,
 AVG(Age %AFTERHAVING) AS AvgMiddleAge
 FROM Sample.Person
 WHERE Home_State IN ('ME','NH','VT','MA','RI','CT','NY')
 HAVING Age > 40
 ORDER BY Home_State
```

GROUP BY, HAVING, and ORDER BY

Here, the **GROUP BY** clause causes the query to compute the AvgAge column for each Home_State group. The **GROUP BY** clause also limits the output display to the first record encountered from each Home_State. The **HAVING** clause governs the AvgMiddleAge computed column, calculating the average age of those over 40 in each Home_State group. The **ORDER BY** clause sequences the display of the rows alphabetically by the Home_State column value.

SQL

```
SELECT Name,Home_State,Age,AVG(Age) AS AvgAge,
 AVG(Age %AFTERHAVING) AS AvgMiddleAge
 FROM Sample.Person
 GROUP BY Home_State
 HAVING Age > 40
 ORDER BY Home_State
```

WHERE, GROUP BY, HAVING, and ORDER BY

In this query, the **WHERE** clause limits the selection to the seven specified northeastern states. The **GROUP BY** clause causes the query to compute the AvgAge column separately for each of these seven Home_State groups. The **GROUP BY** clause also limits the output display to the first record encountered from each specified Home_State. The **HAVING** clause governs the AvgMiddleAge computed column, calculating the average age of those over 40 in each of the seven

Home_State groups. The **ORDER BY** clause sequences the display of the rows alphabetically by the Home_State column value.

SQL

```
SELECT Name,Home_State,Age,AVG(Age) AS AvgAge,
 AVG(Age %AFTERHAVING) AS AvgMiddleAge
 FROM Sample.Person
 WHERE Home_State IN ('ME','NH','VT','MA','RI','CT','NY')
 GROUP BY Home_State
 HAVING Age > 40
 ORDER BY Home_State
```

Select Data from Within ObjectScript Programs Using Embedded SQL and Dynamic SQL

You can use Embedded SQL and Dynamic SQL to issue a **SELECT** query from within an ObjectScript program.

The following Embedded SQL program retrieves data values from one record and places them in the output host variables specified in the INTO clause.

ObjectScript

```
  NEW SQLCODE,%ROWCOUNT
  &sql(SELECT Home_State,Name,Age
       INTO :a, :b, :c
       FROM Sample.Person)
  IF SQLCODE=0 {
    WRITE !,"  Name=",b
    WRITE !,"  Age=",c
    WRITE !,"  Home Home_State=",a
    WRITE !,"Row count is: ",%ROWCOUNT }
  ELSE {
    WRITE !,"SELECT failed, SQLCODE=",SQLCODE  }
```

This program retrieves at most one row, so the %ROWCOUNT variable is set to either 0 or 1. To retrieve multiple rows, you must declare a cursor and use the FETCH command. For more details, see Embedded SQL.

The following Dynamic SQL example first tests whether the desired table exists and checks the current user's **SELECT** privilege for that table. It then executes the query and returns a result set. It uses the **WHILE** loop to repeatedly invoke the **%Next** method for the first 10 records of the result set. It displays three column values using **%GetData** methods that specify the column position as specified in the **SELECT** statement:

ObjectScript

```
  SET tname="Sample.Person"
  IF $SYSTEM.SQL.Schema.TableExists(tname)
     & $SYSTEM.SQL.Security.CheckPrivilege($USERNAME,"1,"_tname,"s")
     {GOTO SpecifyQuery}
  ELSE {WRITE "Table unavailable"  QUIT}
SpecifyQuery
  SET myquery = 3
  SET myquery(1) = "SELECT Home_State,Name,SSN,Age"
  SET myquery(2) = "FROM "_tname
  SET myquery(3) = "ORDER BY Name"
  SET tStatement = ##class(%SQL.Statement).%New()
  SET qStatus = tStatement.%Prepare(.myquery)
    IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET rset = tStatement.%Execute()
  IF rset.%SQLCODE=0 {
    SET x=0
    WHILE x < 10 {
     SET x=x+1
     SET status=rset.%Next()
     WRITE rset.%GetData(2)," "   /* Name column */
     WRITE rset.%GetData(1)," "   /* Home_State column */
     WRITE rset.%GetData(4),!     /* Age column */
    }
    WRITE !,"End of Data"
    WRITE !,"SQLCODE=",rset.%SQLCODE," Row Count=",rset.%ROWCOUNT
  }
  ELSE {
    WRITE !,"SELECT failed, SQLCODE=",rset.%SQLCODE }
```

For more details, see Dynamic SQL.

Change Case of Columns in Result Set

Column names specified in *selectItem* are not case-sensitive. However, unless you supply a column alias, the name of a column in the result set follows the letter case of the SqlFieldName associated with the column property. The letter case of the SqlFieldName corresponds to the column name as specified in the table definition, not as specified in the *selectItem* list. Therefore, SELECT name FROM Sample.Person returns the column label as Name. Using a column alias allows you to specify the letter case to display. For example, this query displays the Name column in the result set as NAME (all caps).

SQL

```
SELECT name AS NAME
FROM Sample.Person
```

Letter case resolution takes time. To maximize **SELECT** performance, specify the exact letter case of the column name as specified in the table definition. However, determining the exact letter case of a column in the table definition is often inconvenient and prone to error. Instead, you can use a column alias to avoid letter case issues. Note that all references to the column alias must match in letter case.

The following Dynamic SQL example requires letter case resolution (the SqlFieldNames are "Latitude" and "Longitude"):

ObjectScript

```
  set query = "SELECT latitude,longitude FROM Sample.USZipCode"
  set statement = ##class(%SQL.Statement).%New()

  set status = statement.%Prepare(query)
  if $$$ISERR(status) {write "%Prepare failed:" do $SYSTEM.Status.DisplayError(status) quit}

  set rset = statement.%Execute()
  if (rset.%SQLCODE '= 0) {write "%Execute failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message
 quit}

  while rset.%Next()
  {
    write rset.latitude," ",rset.longitude,!
  }
  if (rset.%SQLCODE < 0) {write "%Next failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message
quit}
```

The following Dynamic SQL example does not require letter case resolution and therefore executes faster:

ObjectScript

```
  set query = "SELECT latitude AS northsouth,longitude AS eastwest FROM Sample.USZipCode"
  set statement = ##class(%SQL.Statement).%New()

  set status = statement.%Prepare(query)
  if $$$ISERR(status) {write "%Prepare failed:" do $SYSTEM.Status.DisplayError(status) quit}

  set rset = statement.%Execute()
  if (rset.%SQLCODE '= 0) {write "%Execute failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message
 quit}

  while rset.%Next()
  {
    write rset.northsouth," ",rset.eastwest,!
  }
  if (rset.%SQLCODE < 0) {write "%Next failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message
quit}
```

Distinguish Between Column Names in Multi-Table Queries

The returned result set of **SELECT** queries do not include the table alias prefix, *tableAlias*. Therefore, this query returns two columns named Name:

SQL

```
SELECT p.Name,e.Name
FROM Sample.Person AS p LEFT JOIN Sample.Employee AS e ON p.Name=e.Name
```

To distinguish the columns in such queries, specify column aliases. For example, this revised query returns the two columns as `PersonName` and `EmployeeName`:

SQL

```
SELECT p.Name AS PersonName,e.Name AS EmployeeName
FROM Sample.Person AS p LEFT JOIN Sample.Employee AS e ON p.Name=e.Name
```

Security and Privileges

To perform a **SELECT** query on one or more tables, you must have one or more of the following:

- Column-level **SELECT** privileges for all of the specified *selectItem* columns

- Table-level **SELECT** privileges for the specified *table* tables or views

- **SELECT** privileges on the schema of the table

A *selectItem* column specified using a table alias (such as `t.Name` or `"MyAlias".Name`) requires only column-level **SELECT** privileges, not table-level **SELECT** privileges.

When using **SELECT ***, column-level privileges cover all table columns named in the **GRANT** statement. Table-level privileges cover all table columns, including columns added after the privilege assignment.

Failing to have the necessary privileges results in an SQLCODE -99 error (Privilege Violation). To determine if the current user has **SELECT** privilege by invoking the %CHECKPRIV command. You can determine if a specified user has table-level **SELECT** privilege by invoking the **$SYSTEM.SQL.Security.CheckPrivilege()** method. For more on privilege assignment, see GRANT.

Note: Having table-level **SELECT** privileges for a table is not a sufficient test that the table actually exists. If the specified user has the `%All` role, then **CheckPrivilege()** returns 1 even if the specified table or view does not exist.

A **SELECT** query that does not have a **FROM** clause does not require any **SELECT** privileges. A **SELECT** query that contains a **FROM** clause requires **SELECT** privileges, even if no column data is accessed by the query.

More About

SELECT Status and Return Values

When you perform a **SELECT** operation, InterSystems IRIS sets a status variable, SQLCODE, that indicates the success or failure of the operation. In addition, the **SELECT** operation sets the %ROWCOUNT local variable to the number of selected rows. Successful completion of a **SELECT** operation generally sets SQLCODE=0 and `%ROWCOUNT` to the number of rows selected. If embedded SQL code contains a simple **SELECT** statement, data from (at most) one row is selected, so SQLCODE=0 and `%ROWCOUNT` is set to either 0 or 1. If an embedded SQL **SELECT** statement declares a cursor and fetches data from multiple rows, the operation completes when the cursor advances to the end of the data (SQLCODE=100). At that point, `%ROWCOUNT` is set to the total number of rows selected. For more details, see FETCH.

The values returned from a **SELECT** query are known as a *result set*. In Dynamic SQL, **SELECT** retrieves values into the %SQL.Statement class. For more details, see Dynamic SQL and the %SQL.Statement class reference page.

SELECT can also be used to return a value from an SQL function, a host variable, or a literal. A **SELECT** query can combine returning these non-database values with retrieving values from tables or views. When a **SELECT** query returns *only* non-database values, the FROM clause is optional. For more details, see FROM.

Sharding

Sharding is transparent to SQL queries, and no special query syntax is required. A query does not need to know whether a table specified in the **FROM** clause is sharded or non-sharded. The same query can access sharded and non-sharded tables. A query can include joins between sharded and non-sharded tables.

A sharded table is defined using the **CREATE TABLE** command. It must be defined in the master namespace on the shard master data server. This master namespace can also include non-sharded tables.

Transaction Processing

A transaction performing a query is defined as either READ COMMITTED or READ UNCOMMITTED. The default is READ UNCOMMITTED. A query that is not in a transaction is defined as READ UNCOMMITTED.

- In READ UNCOMMITTED mode, a **SELECT** statement returns the current state of the data, including changes made to the data by transactions in progress that have not been committed. These changes can be subsequently rolled back.

- In READ COMMITTED mode, the behavior depends on the contents of the **SELECT** statement. Normally, a **SELECT** statement in read committed mode returns only insert and update changes to data that has been committed. Data rows deleted by a transaction in progress are not returned, even though these deletes have not been committed and can be rolled back.

  However, if the **SELECT** statement contains a `%NOLOCK` keyword, a DISTINCT clause, or a GROUP BY clause, the **SELECT** query returns the current state of the data, including changes made to data during the current transaction that have not been committed. An aggregate function in a **SELECT** statement also returns the current state of the data for the specified columns, including uncommitted changes.

For more details, see SET TRANSACTION and START TRANSACTION.

See Also

- SELECT clauses: DISTINCT, FROM, GROUP BY, HAVING, INTO, ORDER BY, TOP, WHERE

- JOIN, UNION

- CREATE VIEW

- CREATE TABLE, ALTER TABLE, DROP TABLE

- CREATE QUERY, DROP QUERY

- INSERT, INSERT OR UPDATE, UPDATE, DELETE

- Querying the Database

- SQL and Object Settings Pages

- SQLCODE error messages

SET ML CONFIGURATION (SQL)

Sets an ML configuration as the default.

Synopsis

```
SET ML CONFIGURATION ml-configuration-name
```

Arguments

ml-configuration-name	The name of the ML configuration.

Description

The **SET ML CONFIGURATION** command sets the specified ML configuration as the system default for all ensuing **TRAIN MODEL** statements. Only one ML configuration can be set as system default by each **SET ML CONFIGURATION** statement.

Required Security Privileges

Calling **SET ML CONFIGURATION** requires a USE object privilege; otherwise, there is a SQLCODE –99 error (Privilege Violation). You can determine if the current user has USE privilege by invoking the %CHECKPRIV command or the $SYSTEM.SQL.Security.CheckPrivilege() method.

Examples

```
CREATE MODEL H2OMODEL PREDICTING (label) FROM data
SET ML CONFIGURATION %H2O
TRAIN MODEL H2OMODEL
```

See Also

- ALTER ML CONFIGURATION, CREATE ML CONFIGURATION

SET OPTION (SQL)

Sets an execution option.

Synopsis

```
SET OPTION option_keyword = value
```

Description

The **SET OPTION** statement is used to set execution options, such as the compile mode, SQL configuration settings, and the locale settings governing date, time, and numeric conventions. Only one keyword option can be set by each **SET OPTION** statement.

SET OPTION supports the following options:

- AUTO_PARALLEL_THRESHOLD

- COMPILEMODE

- DEFAULT_SCHEMA

- EXACT_DISTINCT

- LOCK_ESCALATION_THRESHOLD

- LOCK_TIMEOUT

- PKEY_IS_IDKEY

- SUPPORT_DELIMITED_IDENTIFIERS

- Locale Options (date, time, and numeric conventions)

SET OPTION can be used in Dynamic SQL (including the SQL Shell) and in Embedded SQL.

Other **SET OPTION** arguments (not documented here) are parsed by InterSystems IRIS for SQL compatibility, but perform no operation.

Because **SET OPTION** prepares and executes quickly, and is generally run only once, InterSystems IRIS does not create a cached query for **SET OPTION** in ODBC, JDBC, or Dynamic SQL.

The following options are supported by InterSystems IRIS:

AUTO_PARALLEL_THRESHOLD

The AUTO_PARALLEL_THRESHOLD option is set to an integer *n* that determines whether parallel processing should be applied to a query when automatic parallel processing is enabled. Because there are performance costs associated with parallel processing, a threshold needs to be established for when parallel processing is advantageous. The higher *n* is, the lower the chance that an InterSystems SQL query executes using parallel processing. The default is 3200. This is a system-wide setting. The value *n* corresponds roughly to the minimal number of tuples needed in the visited map for parallel processing to occur.

When `AutoParallel` is disabled, the AUTO_PARALLEL_THRESHOLD option has no effect.

This option can also be set using the **$SYSTEM.SQL.Util.SetOption()** method `AutoParallelThreshold` option.

For further details, refer to AutoParallelThreshold.

COMPILEMODE

The COMPILEMODE option sets the compile mode to DEFERRED, IMMEDIATE, INSTALL, or NOCHECK for the current namespace. The default is IMMEDIATE. Changing from DEFERRED to IMMEDIATE compile mode causes any classes in the Deferred Compile Queue to be compiled immediately. If all class compilations are successful, InterSystems IRIS sets SQLCODE to 0. If there are any errors, SQLCODE is set to -400. Class compilation errors are logged in the ^mtemp2 ("Deferred Compile Mode","Error"). If SQLCODE is set to -400, you should view this global structure for more precise error messages. The INSTALL compile mode is similar to the DEFERRED compile mode, but it should only be used for DDL installations where there is no data in the tables.

The NOCHECK compile mode is similar to IMMEDIATE, except that it skips checking of the following constraints when compiling: If a table is dropped, InterSystems IRIS does not check foreign key constraints in other tables that reference the dropped table. If a foreign key constraint is added, InterSystems IRIS does not check existing data to ensure that it is valid for this foreign key. If a NOT NULL constraint is added, InterSystems IRIS does not check existing data for NULLs or assign the field's default value. If a UNIQUE or Primary Key constraint is deleted, InterSystems IRIS does not check if a foreign key in this table or another table references the dropped key.

This option can also be set using the **$SYSTEM.SQL.Util.SetOption()** method `CompileMode` options.

DEFAULT_SCHEMA

The DEFAULT_SCHEMA option sets the default schema system-wide for all namespaces. This default remains in effect until explicitly changed. The default schema name is used to supply a schema name for all unqualified table, view, or stored procedure names.

You can specify a literal schema name or specify _CURRENT_USER. If you specify _CURRENT_USER as the default schema name, InterSystems IRIS assigns the user name of the currently logged-in process as the default schema name. For further details, refer to Schema Name.

EXACT_DISTINCT

The EXACT_DISTINCT boolean option specifies whether DISTINCT processing (TRUE) or Fast Distinct processing (FALSE) should be used system-wide. The system-wide default is to use Fast Distinct processing.

When EXACT_DISTINCT=TRUE, GROUP BY and DISTINCT queries produce original values. When EXACT_DISTINCT=FALSE, Fast Distinct is enabled, causing SQL queries involving DISTINCT or GROUP BY clauses to run more efficiently by making better use of indexes (if indexes are available). However, the values returned by such queries are collated in the same way they are stored within the index. This means the results of such queries may be all uppercase. This may have an effect on case-sensitive applications.

This option can also be set using the **$SYSTEM.SQL.Util.SetOption()** method `FastDistinct` boolean option.

For further details, refer to FastDistinct.

LOCK_ESCALATION_THRESHOLD

The LOCK_ESCALATION_THRESHOLD option is set to an integer n that determines when to escalate row locking to table locking. The default is 1000. The value n is the number of inserts, updates, or deletes for a single table within a single transaction that will trigger a table-level lock when reached. This is a system-wide setting for all namespaces. For example, if the lock threshold is 1000 and a process starts a transaction and then inserts 2000 rows, after the 1001st row is inserted the process will attempt to acquire a table-level lock instead of continue to lock individual rows. This is to help keep the lock table from becoming too full.

This option can also be set using the **$SYSTEM.SQL.Util.SetOption()** method `LockThreshold` option.

For further details, see Modify Transaction Lock Threshold.

LOCK_TIMEOUT

The LOCK_TIMEOUT numeric option lets you set the default lock timeout for the current process. The LOCK_TIMEOUT *value* is the number of seconds to wait when trying to establish a lock during SQL execution. This lock timeout is used when a locking conflict prevents the current process from immediately locking a record, table, or other entity for a **LOCK**, **INSERT**, **UPDATE**, **DELETE**, or **SELECT** operation. InterSystems SQL continues to try to establish the lock until the timeout expires, at which point an SQLCODE -110 or -114 error is generated.

Available values are positive integers and zero. The timeout setting is per process. You can determine the lock timeout setting for the current process using the **$SYSTEM.SQL.Util.GetOption("ProcessLockTimeout")** method.

If you do not set the lock timeout for the current process, it defaults to the current system-wide lock timeout setting. If your ODBC connection disconnects and reconnects, the reconnected process uses the current system-wide lock timeout setting. The default system-wide lock timeout is 10 seconds.

For further details on locking conflicts and per-process and system-wide SQL lock timeout settings, refer to the LOCK command.

PKEY_IS_IDKEY

The PKEY_IS_IDKEY boolean option specifies whether primary keys are also ID keys system-wide. Available values are TRUE and FALSE. If TRUE, and the field does not contain data, the primary key is created as an ID key. That is, the primary key of the table also becomes the IDKey index in the class definition. If the field *does* contain data, the IDKey index is not defined. If the primary key is defined as the IDKey index, data access is more efficient, but a primary key value, once set, can never be modified. Once set, you cannot change the value assigned to a primary key, nor can you assign a different key as the primary key. Use of this option also changes the primary key collation default; primary key string values default to EXACT collation. If FALSE, the primary key and ID key are defined as independent, which is less efficient. However, primary key values are modifiable, and primary key string values default to the current collation type default, which is SQLUPPER by default.

To set the PKEY_IS_IDKEY option, you must have the %Admin_Manage:USE privilege. Otherwise, you receive an SQLCODE -99 error (Privilege Violation). Once set, this option takes effect system-wide for all processes. The system-wide default for this option can also be set using:

- The system-wide **$SYSTEM.SQL.Util.SetOption()** method configuration option `DDLPKeyNotIDKey`. To determine the current setting, call **$SYSTEM.SQL.CurrentSettings()** which displays `Are primary keys created through DDL not ID keys`; the default is 1.

- A Management Portal configuration setting. Select **System Administration**, **Configuration**, **SQL and Object Settings**, **SQL**. View or modify the current setting of **Define primary key as ID key for tables created via DDL**.

The PKEY_IS_IDKEY setting remains in effect until reset through another **SET OPTION** PKEY_IS_IDKEY or until the InterSystems IRIS Configuration is reactivated, which resets this parameter to the InterSystems IRIS System Configuration setting.

SUPPORT_DELIMITED_IDENTIFIERS

By default, delimited identifiers are supported system-wide. The SUPPORT_DELIMITED_IDENTIFIERS boolean option allows you to change support for delimited identifiers system-wide. Available values are TRUE and FALSE. If TRUE, a string delimited by double quotation marks is considered an identifier within an SQL statement. If FALSE, a string delimited by double quotation marks is considered a string literal within an SQL statement.

To set the SUPPORT_DELIMITED_IDENTIFIERS option, you must have the %Admin_Manage:USE privilege. Otherwise, you receive an SQLCODE -99 error (Privilege Violation). Once set, this option takes effect system-wide for all processes. The SUPPORT_DELIMITED_IDENTIFIERS setting remains in effect until reset through another **SET OPTION** SUP-PORT_DELIMITED_IDENTIFIERS, or until changed system-wide by the **$SYSTEM.SQL.Util.SetOption()** method `DelimitedIdentifiers` option.

To determine the current setting, call **$SYSTEM.SQL.CurrentSettings()**.

Locale Options

Locale options are keyword options used to set your InterSystems IRIS Locale settings for date, time, and numeric conventions for the current process. The available keyword options are AM, DATE_FORMAT, DATE_MAXIMUM, DATE_MINIMUM, DATE_SEPARATOR, DECIMAL_SEPARATOR, MIDNIGHT, MINUS_SIGN, MONTH_ABBR, MONTH_NAME, NOON, NUMERIC_GROUP_SEPARATOR, NUMERIC_GROUP_SIZE, PM, PLUS_SIGN, TIME_FORMAT, TIME_PRECISION, TIME_SEPARATOR, WEEKDAY_ABBR, WEEKDAY_NAME, and YEAR_OPTION. All of these options can be set to a literal, and all take a default (American English conventions). The TIME_PRECISION option is configurable (see below). If you set any of these options to an invalid value, InterSystems IRIS issues an SQLCODE -129 error (Illegal value for **SET OPTION** locale property). See the ObjectScript $ZDATETIME function for an explanation of date and time formats and options.

Date/Time Option Keyword	Description
AM	String. Default is 'AM'
DATE_FORMAT	Integer. Default is 1. Available values are 0 through 15. For an explanation of these date formats, see the ObjectScript $ZDATE function.
DATE_MAXIMUM	Integer. Default is 2980013 (12/31/9999). Can be set to an earlier date, but not to a later date.
DATE_MINIMUM	Positive Integer. Default is 0 (12/31/1840). Can be set to a later date, but not to an earlier date.
DATE_SEPARATOR	Character. Default is '/'
DECIMAL_SEPARATOR	Character. Default is '.'
MIDNIGHT	String. Default is 'MIDNIGHT'
MINUS_SIGN	Character. Default is '-'
MONTH_ABBR	String. Default is ' Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec'. (Note that this string begins with a space character, which is the default separator character.)
MONTH_NAME	String. Default is ' January February March April May June ... November December'. (Note that this string begins with a space character, which is the default separator character.)
NOON	String. Default is 'NOON'
NUMERIC_GROUP_SEPARATOR	Character. Default is ','
NUMERIC_GROUP_SIZE	Integer. Default is 3.
PM	String. Default is 'PM'
PLUS_SIGN	Character. Default is '+'
TIME_FORMAT	Integer. Default is 1. Available values are 1 through 4. For an explanation of these time formats, see the ObjectScript $ZTIME function.
TIME_PRECISION	Integer from 0 through 9 (inclusive). Default is 0. The number of digits of fractional seconds. Configurable, as described below.

Date/Time Option Keyword	Description
TIME_SEPARATOR	Character. Default is ':'
WEEKDAY_ABBR	String. Default is ' Sun Mon Tue Wed Thu Fri Sat'. (Note that this string begins with a space character, which is the default separator character.)
WEEKDAY_NAME	String. Default is ' Sunday Monday Tuesday Wednesday Thursday Friday Saturday'. (Note that this string begins with a space character, which is the default separator character.)
YEAR_OPTION	Integer. Default is 0. Available values are 0 through 6. For an explanation of these ways of representing 2-digit and 4-digit years, see the ObjectScript $ZDATE function.

To configure TIME_PRECISION system-wide, go to the Management Portal, select **System Administration**, **Configuration**, **SQL and Object Settings**, **SQL**. View and edit the current setting of **Default time precision for GETDATE(), CURRENT_TIME, and CURRENT_TIMESTAMP**. This specifies the number of digits of precision for fractional seconds. The default is 0. The range of allowed values is 0 through 9 digits of precision. The actual number of meaningful digits of fractional seconds is platform-dependent.

See Also

- SQL date and time functions: CURRENT_TIMESTAMP, DATEPART, DATENAME, GETDATE, NOW

- SQL date functions: DAYNAME, DAYOFWEEK, DAYOFMONTH, DAYOFYEAR, WEEK, MONTH, MONTH-NAME, QUARTER, YEAR, CURDATE, CURRENT_DATE, TO_DATE

- SQL time functions: HOUR, MINUTE, SECOND, CURTIME, CURRENT_TIME

- SQL and Object Settings Pages

- SQLCODE error messages

- ObjectScript functions: $ZDATE $ZDATETIME $ZTIME

SET TRANSACTION (SQL)

Sets parameters for transactions.

Synopsis

```
SET TRANSACTION [%COMMITMODE commitmode]
SET TRANSACTION [transactionmodes]
```

Description

A **SET TRANSACTION** statement sets parameters that govern SQL transactions for the current process. These parameters take effect at the beginning of the next transaction and continue in effect for the duration of the current process or until explicitly reset. They do not automatically reset to defaults at the end of a transaction.

A single **SET TRANSACTION** statement can be used to set either the *commitmode* parameter or the *transactionmodes* parameters, but not both.

The same parameters can be set using the **START TRANSACTION** command, which can both set parameters and begin a new transaction. The parameters can also be set using method calls.

SET TRANSACTION does not begin a transaction, and therefore does not increment the $TLEVEL transaction level counter.

SET TRANSACTION can be used in Dynamic SQL (including the SQL Shell) and in Embedded SQL.

%COMMITMODE

The %COMMITMODE keyword allows you to specify whether automatic transaction commitment is performed. The available options are:

- IMPLICIT: automatic transaction commitment is on (the default). SQL automatically initiates a transaction when a program issues a database modification operation (**INSERT**, **UPDATE**, or **DELETE**). The transaction continues until either the operation completes successfully and SQL automatically commits the changes, or the operation is unable to complete successfully on all rows and SQL automatically rolls back the entire operation. Each database operation (**INSERT**, **UPDATE**, or **DELETE**) constitutes a separate transaction. Successful completion of the database operation automatically clears the rollback journal, releases locks, and decrements $TLEVEL. No **COMMIT** statement is needed. This is the default setting.

- EXPLICIT: automatic transaction commitment is off. SQL automatically initiates a transaction when a program issues the first database modification operation (**INSERT**, **UPDATE**, or **DELETE**). This transaction continues until it is explicitly concluded. Upon successful completion you issue a **COMMIT** statement. If a database modification operation fails you issue a **ROLLBACK** statement to revert the database to the point prior to the beginning of the transaction. In EXPLICIT mode the number of database operations per transaction is user-defined.

- NONE: no automatic transaction processing. A transaction is not initiated unless explicitly invoked by a **START TRANSACTION** statement. The transaction must be explicitly concluded by issuing either a **COMMIT** or **ROLLBACK** statement. Thus whether a database operation is included in a transaction, and the number of database operations in a transaction are both user-defined.

TRUNCATE TABLE does not occur within an automatically initiated transaction. If journaling and rollback of **TRUNCATE TABLE** is required, you must explicitly specify a **START TRANSACTION** and conclude with an explicit **COMMIT** or **ROLLBACK**.

You can determine the %COMMITMODE setting for the current process using the **GetOption("AutoCommit")** method, as shown in the following ObjectScript example:

ObjectScript

```
SET stat=$SYSTEM.SQL.Util.SetOption("AutoCommit",$RANDOM(3),.oldval)
IF stat'=1 {WRITE "SetOption failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET x=$SYSTEM.SQL.Util.GetOption("AutoCommit")
IF x=1 {
  WRITE "%COMMITMODE IMPLICIT (default behavior):",!,
        "each database operation is a separate transaction",!,
        "with automatic commit or rollback" }
ELSEIF x=0 {
  WRITE "%COMMITMODE NONE:",!,
        "No automatic transaction support",!,
        "You must use START TRANSACTION to start a transaction",!,
        "and COMMIT or ROLLBACK to conclude one" }
ELSE {
  WRITE "%COMMITMODE EXPLICIT:",!,
        "the first database operation automatically",!,
        "starts a transaction; to end the transaction",!,
        "explicit COMMIT or ROLLBACK required" }
```

The %COMMITMODE can be set in ObjectScript using the **SetOption()** method, as follows `SET status=$SYSTEM.SQL.Util.SetOption("AutoCommit",intval,.oldval)`. The available method values are 0 (NONE), 1 (IMPLICIT), and 2 (EXPLICIT).

ISOLATION LEVEL

You specify an ISOLATION LEVEL for a process that is issuing a query. The ISOLATION LEVEL options permit you to specify whether changes that are in progress should be available for read access by the query. If another concurrent process is performing inserts or updates to a table and those changes to the table are in a transaction, those changes are in progress, and could, potentially, be rolled back. By setting the ISOLATION LEVEL for your process that is querying that table, you can specify whether you wish to include or exclude these changes in progress from the query results.

- READ UNCOMMITTED states that all changes are immediately available for query access. This includes changes that may subsequently be rolled back. READ UNCOMMITTED ensures that your query will return results without waiting for a concurrent insert or update process, and will not fail due to a lock timeout error. However, the results of a READ UNCOMMITTED may include values that are not committed; these values may be internally inconsistent because the insert or update operation has only partially completed, and these values may be subsequently rolled back. READ UNCOMMITTED is the default if your query process is not in an explicit transaction, or if the transaction does not specify an ISOLATION LEVEL. READ UNCOMMITTED is incompatible with READ WRITE access; attempting to specify both in the same statement results in an SQLCODE -92 error.

- READ VERIFIED states that uncommitted data from other transactions is immediately available, and no locking is performed. This includes changes that may subsequently be rolled back. However, unlike READ UNCOMMITTED, a READ VERIFIED transaction will re-check any conditions that could be invalidated by uncommitted or newly committed data which would result in output that does not satisfy the query conditions. Because of this condition re-check, READ VERIFIED is more accurate but less efficient than READ UNCOMMITTED and should only be used when concurrent updates to the data being checked by the conditions is likely to occur. READ VERIFIED is incompatible with READ WRITE access; attempting to specify both in the same statement results in an SQLCODE -92 error.

- READ COMMITTED states that only those changes that have been committed are available for query access. This ensures that a query is performed on the database in a consistent state, not while a group of changes are being made, a group of changes which may be subsequently rolled back. If requested data has been changed, but the changes have not been committed (or rolled back), the query waits for transaction completion. If a lock timeout occurs while waiting for this data to be available, an SQLCODE -114 error is issued.

READ UNCOMMITTED or READ VERIFIED?

The difference between READ UNCOMMITTED and READ VERIFIED is demonstrated by the following example:

SQL

```
SELECT Name,SSN FROM Sample.Person WHERE Name >= 'M'
```

The query optimizer may choose first to collect all RowID's containing Names meeting the >= 'M' condition from a Name index. Once collected, the Person table is accessed one RowID at a time to retrieve the Name and SSN fields for output. A concurrently running updating transaction could change the Name field of a Person with RowID 72 from 'Smith' to 'Abel' in-between the query's collection of RowID's from the index and its row-by-row access to the table. In this case, the collection of RowID's from the index would contain the RowID for a row that no longer conforms to the Name >= 'M' condition.

READ UNCOMMITTED query processing assumes that the Name >= 'M' condition has been satisfied by the index, and will output whatever Name is present in the table for each RowID it collected from the index. In this example it would therefore output a row with a Name of 'Abel', which does not satisfy the condition.

READ VERIFIED query processing notes that it is retrieving a field from a table for output (Name) that participates in a condition which should have been previously satisfied by the index, and re-checks the condition in case the field value has changed since the index was examined. Upon re-check, it notes that the row no longer satisfies the condition and omits it from the output. Only values that are needed for output have their conditions re-checked: `SELECT SSN FROM Person WHERE Name >= 'M'` would output the row with RowID 72 in this example.

Exceptions to READ COMMITTED

When ISOLATION LEVEL read committed is in effect, either through setting ISOLATION LEVEL READ COMMITTED or the **SetOption()** method, as follows SET `status=$SYSTEM.SQL.Util.SetOption("IsolationMode",1,.oldval)`. SQL can retrieve only those changes to the data that have been committed. However, there are significant exceptions to this rule:

- A deleted row is never returned by a query, even when the transaction that deleted the row is in progress and the delete may be subsequently rolled back. ISOLATION LEVEL READ COMMITTED ensures that inserts and updates are in a consistent state, but not deletes.

- If you query contains an aggregate function, the aggregate result returns the current state of the data, regardless of the specified ISOLATION LEVEL. Therefore, inserts and updates are in progress (and may subsequently be rolled back) are included in aggregate results. Deletes that are in progress (and may subsequently be rolled back) are *not* included in aggregate results. This is because an aggregate operation requires access to data from many rows of a table.

- A **SELECT** query that contains a DISTINCT clause or a GROUP BY clause is unaffected by the ISOLATION LEVEL setting. A query containing one of these clauses returns the current state of the data, including changes in progress that may be subsequently rolled back. This is because these query operations require access to data from many rows of a table.

- A query with the %NOLOCK keyword.

Note: On InterSystems IRIS implementations with ECP (Enterprise Cache Protocol) use of READ COMMITTED may result in significantly slower performance when compared to READ UNCOMMITTED. Developers should weigh the superior performance of READ UNCOMMITTED against the greater data accuracy of READ COMMITTED when defining transactions that involve ECP.

For further details, refer to Transaction Processing.

ISOLATION LEVEL in Effect

You can set the ISOLATION LEVEL for a process using **SET TRANSACTION** (without starting a transaction), **START TRANSACTION** (setting isolation mode and starting a transaction), or a **SetOption("IsolationMode")** method call.

The specified ISOLATION LEVEL remains in effect until explicitly reset by a **SET TRANSACTION**, **START TRANSACTION**, or a **SetOption("IsolationMode")** method call. Because **COMMIT** or **ROLLBACK** is only meaningful for changes to the data, not data queries, a **COMMIT** or **ROLLBACK** operation has no effect on the ISOLATION LEVEL setting.

The ISOLATION LEVEL in effect at the start of a query remains in effect for the duration of the query.

You can determine the ISOLATION LEVEL for the current process using the **GetOption("IsolationMode")** method call. You can also set the isolation mode for the current process using the **SetOption("IsolationMode")** method call. These methods specify READ UNCOMMITTED (the default) as 0, READ COMMITTED as 1, and READ VERIFIED as 3. Specifying any other numeric value leaves the isolation mode unchanged. No error or change occurs if you set the isolation mode to the current isolation mode. Use of these methods is shown in the following example:

ObjectScript

```
WRITE $SYSTEM.SQL.Util.GetOption("IsolationMode")," default",!
&sql(START TRANSACTION ISOLATION LEVEL READ COMMITTED,READ WRITE)
WRITE $SYSTEM.SQL.Util.GetOption("IsolationMode")," after START TRANSACTION",!
DO $SYSTEM.SQL.Util.SetOption("IsolationMode",0,.stat)
IF stat=1 {
  WRITE $SYSTEM.SQL.Util.GetOption("IsolationMode")," after IsolationMode=0 call",! }
ELSE { WRITE "Set IsolationMode error" }
&sql(COMMIT)
```

The isolation mode and the access mode must always be compatible. Changing the access mode changes the isolation mode, as shown in the following example:

ObjectScript

```
WRITE $SYSTEM.SQL.Util.GetOption("IsolationMode")," default",!
&sql(SET TRANSACTION ISOLATION LEVEL READ COMMITTED,READ WRITE)
WRITE $SYSTEM.SQL.Util.GetOption("IsolationMode")," after SET TRANSACTION",!
&sql(START TRANSACTION READ ONLY)
WRITE $SYSTEM.SQL.Util.GetOption("IsolationMode")," after changing access mode",!
&sql(COMMIT)
```

Arguments

%COMMITMODE commitmode

An optional argument specifying the manner in which transactions are committed to the database. Available values are EXPLICIT, IMPLICIT, and NONE. The default is IMPLICIT.

transactionmodes

An optional argument that specifies the isolation mode and access mode for the transaction. You can specify a value for either an isolation mode, an access mode, or for both modes as a comma-separated list.

Valid values for isolation mode are ISOLATION LEVEL READ COMMITTED, ISOLATION LEVEL READ UNCOMMITTED, and ISOLATION LEVEL READ VERIFIED. The default is ISOLATION LEVEL READ UNCOMMITTED.

Valid values for access mode are READ ONLY and READ WRITE. Note that only ISOLATION LEVEL READ COMMITTED is compatible with access mode READ WRITE.

Examples

The following Embedded SQL example uses two **SET TRANSACTION** statements to set transaction parameters. Note that **SET TRANSACTION** does not increment the transaction level (*$TLEVEL*). The **START TRANSACTION** command initiates a transaction and increments *$TLEVEL*:

ObjectScript

```
&sql(SET TRANSACTION %COMMITMODE EXPLICIT)
  WRITE !,"Set transaction commit mode, SQLCODE=",SQLCODE
  WRITE !,"Transaction level=",$TLEVEL
&sql(SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED)
  WRITE !,"Set transaction isolation mode, SQLCODE=",SQLCODE
  WRITE !,"Transaction level=",$TLEVEL
&sql(START TRANSACTION)
  WRITE !,"Start transaction, SQLCODE=",SQLCODE
  WRITE !,"Transaction level=",$TLEVEL
&sql(SAVEPOINT a)
  WRITE !,"Set Savepoint a, SQLCODE=",SQLCODE
  WRITE !,"Transaction level=",$TLEVEL
&sql(COMMIT)
  WRITE !,"Commit transaction, SQLCODE=",SQLCODE
  WRITE !,"Transaction level=",$TLEVEL
```

See Also

- COMMIT, ROLLBACK, SAVEPOINT, START TRANSACTION, $TLEVEL

- Transaction Processing

START TRANSACTION (SQL)

Begins a transaction.

Synopsis

```
START TRANSACTION [%COMMITMODE commitmode]
START TRANSACTION [transactionmodes]
```

Description

A **START TRANSACTION** statement initiates a transaction. **START TRANSACTION** immediately initiates a transaction, regardless of the current commit mode setting. A transaction beginning with **START TRANSACTION** must be concluded by issuing an explicit **COMMIT** or **ROLLBACK**, regardless of the current commit mode setting.

START TRANSACTION is optional.

- If your process is only querying the data (**SELECT** statements), you can use SET TRANSACTION to establish the ISOLATION LEVEL. A **START TRANSACTION** is not needed.

- If your process is modifying the data, whether you need to explicitly begin an SQL transaction by issuing a **START TRANSACTION** depends on the current commit mode setting for the process (also referred to as the AutoCommit setting). If the commit mode for the current process is IMPLICIT or EXPLICIT, issuing a **START TRANSACTION** is optional. If you omit **START TRANSACTION**, the system automatically initiates a transaction when you invoke a modify data operation (**DELETE**, **UPDATE**, or **INSERT**). If you specify **START TRANSACTION** a transaction is immediately initiated, and must be concluded by an explicit **COMMIT** or **ROLLBACK**.

When **START TRANSACTION** initiates a transaction it increments the $TLEVEL transaction level counter from 0 to 1, indicating a transaction is in progress. You can also determine if a transaction is in progress by checking the SQLCODE set by the %INTRANSACTION statement. Issuing a **START TRANSACTION** when a transaction is in progress has no effect on **$TLEVEL** or **%INTRANSACTION**.

InterSystems SQL does not support nested transactions. Issuing a **START TRANSACTION** when a transaction is already in progress does not initiate another transaction and does not return an error code. InterSystems SQL does support savepoints, allowing a partial rollback of a transaction.

If a transaction is not in progress when you issue a **SAVEPOINT** statement, **SAVEPOINT** initiates a transaction. However, this means of initiating a transaction is not recommended.

An SQLCODE -400 is issued if a transaction operation fails to complete successfully.

Setting Parameters

Optionally, **START TRANSACTION** can be used to set parameters. The parameter settings you specify take effect immediately. However, any transaction initiated with a **START TRANSACTION** must be concluded with an explicit **COMMIT** or **ROLLBACK**, regardless of how you set the *commitmode* parameter. Parameter settings continue in effect for the duration of the current process or until explicitly reset. They do not automatically reset to defaults at the end of a transaction.

A single **START TRANSACTION** statement can be used to set either the *commitmode* parameter or the *transactionmodes* parameters, but not both. To set both, you may issue a **SET TRANSACTION** and a **START TRANSACTION**, or two **START TRANSACTION** statements. Only the first **START TRANSACTION** initiates a transaction.

After issuing a **START TRANSACTION**, you can change these parameter settings during the transaction by issuing another **START TRANSACTION**, a **SET TRANSACTION**, or a method call. Changing the *commitmode* parameter does not remove the requirement to conclude the current transaction with an explicit **COMMIT** or **ROLLBACK**.

You can use the **SET TRANSACTION** statement to set the *commitmode* or *transactionmodes* parameters without starting a transaction. These parameters can also be set using method calls, either outside of a transaction or within a transaction.

%COMMITMODE

The %COMMITMODE keyword allows you to specify automatic transaction initiation and commitment behavior for the current process. A **START TRANSACTION** %COMMITMODE changes the commit mode setting for all future transactions on the current process. It does not affect the transaction initiated by the **START TRANSACTION** statement. Regardless of the current or set commit mode, a **START TRANSACTION** immediately initiates a transaction, and this transaction must be concluded by issuing an explicit **COMMIT** or **ROLLBACK**.

The available %COMMITMODE options are:

- IMPLICIT: automatic transaction commitment is on (the initial process default). SQL automatically initiates a transaction when a program issues a database modification operation (**INSERT**, **UPDATE**, or **DELETE**). The transaction continues until either the operation completes successfully and SQL automatically commits the changes, or the operation is unable to complete successfully on all rows and SQL automatically rolls back the entire operation. Each database operation (**INSERT**, **UPDATE**, or **DELETE**) constitutes a separate transaction. Successful completion of the database operation automatically clears the rollback journal, releases locks, and decrements $TLEVEL. No **COMMIT** statement is needed.

- EXPLICIT: automatic transaction commitment is off. SQL automatically initiates a transaction when a program issues the first database modification operation (**INSERT**, **UPDATE**, or **DELETE**). This transaction continues until it is explicitly concluded. Upon successful completion you issue a **COMMIT** statement. If a database modification operation fails you issue a **ROLLBACK** statement to revert the database to the point prior to the beginning of the transaction. In EXPLICIT mode multiple database modification operations can constitute a single transaction.

- NONE: no automatic transaction processing. Transactions are not initiated unless explicitly invoked by a **START TRANSACTION**. All transactions must be explicitly concluded by issuing either a **COMMIT** or **ROLLBACK** statement. Thus whether a database operation is included in a transaction, and the number of database operations in a transaction are both user-defined.

TRUNCATE TABLE does not occur within an automatically initiated transaction. If journaling and rollback of **TRUNCATE TABLE** is required, you must explicitly specify a **START TRANSACTION** and conclude with an explicit **COMMIT** or **ROLLBACK**.

You can set the %COMMITMODE in ObjectScript using the **SetOption()** method, as follows `SET status=$SYSTEM.SQL.Util.SetOption("AutoCommit",intval,.oldval)`. The available method values are 0 (NONE), 1 (IMPLICIT), and 2 (EXPLICIT).

Note: A sharded table is always in No AutoCommit mode (`SetOption("AutoCommit",0)`), which means all inserts, updates, and deletes to sharded tables are performed outside the scope of a transaction.

ISOLATION LEVEL

You specify an ISOLATION LEVEL for a process that is issuing a query. The ISOLATION LEVEL options permit you to specify whether or not changes that are in progress should be available for read access by the query. If another concurrent process is performing inserts or updates to a table and those changes to the table are in a transaction, those changes are in progress, and could, potentially, be rolled back. By setting the ISOLATION LEVEL for your process that is querying that table, you can specify whether you wish to include or exclude these changes in progress from the query results.

- READ UNCOMMITTED states that all changes are immediately available for query access. This includes changes that may subsequently be rolled back. READ UNCOMMITTED ensures that your query will return results without waiting for a concurrent insert or update process, and will not fail due to a lock timeout error. However, the results of a READ UNCOMMITTED may include values that are not committed; these values may be internally inconsistent because the insert or update operation has only partially completed, and these values may be subsequently rolled back. READ UNCOMMITTED is the default if your query process is not in an explicit transaction, or if the transaction does

not specify an ISOLATION LEVEL. READ UNCOMMITTED is incompatible with READ WRITE access; attempting to specify both in the same statement results in an SQLCODE -92 error.

- READ VERIFIED states that uncommitted data from other transactions is immediately available, and no locking is performed. This includes changes that may subsequently be rolled back. However, unlike READ UNCOMMITTED, a READ VERIFIED transaction will re-check any conditions that could be invalidated by uncommitted or newly committed data which would result in output that does not satisfy the query conditions. Because of this condition re-check, READ VERIFIED is more accurate but less efficient than READ UNCOMMITTED and should only be used when concurrent updates to the data being checked by the conditions is likely to occur. READ VERIFIED is incompatible with READ WRITE access; attempting to specify both in the same statement results in an SQLCODE -92 error.

- READ COMMITTED states that only those changes that have been committed are available for query access. This ensures that a query is performed on the database in a consistent state, not while a group of changes are being made, a group of changes which may be subsequently rolled back. If requested data has been changed, but the changes have not been committed (or rolled back), the query waits for transaction completion. If a lock timeout occurs while waiting for this data to be available, an SQLCODE -114 error is issued.

READ UNCOMMITTED or READ VERIFIED?

The difference between READ UNCOMMITTED and READ VERIFIED is demonstrated by the following example:

SQL

```
SELECT Name,SSN FROM Sample.Person WHERE Name >= 'M'
```

The query optimizer may choose first to collect all RowID's containing Names meeting the >= 'M' condition from a Name index. Once collected, the Person table is accessed one RowID at a time to retrieve the Name and SSN fields for output. A concurrently running updating transaction could change the Name field of a Person with RowID 72 from 'Smith' to 'Abel' in-between the query's collection of RowID's from the index and its row-by-row access to the table. In this case, the collection of RowID's from the index would contain the RowID for a row that no longer conforms to the Name >= 'M' condition.

READ UNCOMMITTED query processing assumes that the Name >= 'M' condition has been satisfied by the index, and will output whatever Name is present in the table for each RowID it collected from the index. In this example it would therefore output a row with a Name of 'Abel', which does not satisfy the condition.

READ VERIFIED query processing notes that it is retrieving a field from a table for output (Name) that participates in a condition which should have been previously satisfied by the index, and re-checks the condition in case the field value has changed since the index was examined. Upon re-check, it notes that the row no longer satisfies the condition and omits it from the output. Only values that are needed for output have their conditions re-checked: `SELECT SSN FROM Person WHERE Name >= 'M'` would output the row with RowID 72 in this example.

Exceptions to READ COMMITTED

When ISOLATION LEVEL read committed is in effect, either through setting ISOLATION LEVEL READ COMMITTED or the **SetOption()** method, as follows `SET status=$SYSTEM.SQL.Util.SetOption("IsolationMode",1,.oldval)`. SQL can retrieve only those changes to the data that have been committed. However, there are significant exceptions to this rule:

- A deleted row is never returned by a query, even when the transaction that deleted the row is in progress and the delete may be subsequently rolled back. ISOLATION LEVEL READ COMMITTED ensures that inserts and updates are in a consistent state, but not deletes.

- If your query contains an aggregate function, the aggregate result returns the current state of the data, regardless of the specified ISOLATION LEVEL. Therefore, inserts and updates in progress (and may subsequently be rolled back) are included in aggregate results. Deletes that are in progress (and may subsequently be rolled back) are *not* included in aggregate results. This is because an aggregate operation requires access to data from many rows of a table.

- A **SELECT** query that contains a DISTINCT clause or a GROUP BY clause is unaffected by the ISOLATION LEVEL setting. A query containing one of these clauses returns the current state of the data, including changes in progress that may be subsequently rolled back. This is because these query operations require access to data from many rows of a table.

- A query with the %NOLOCK keyword.

Note: On InterSystems IRIS implementations with ECP (Enterprise Cache Protocol) use of READ COMMITTED may result in significantly slower performance when compared to READ UNCOMMITTED. Developers should weigh the superior performance of READ UNCOMMITTED against the greater data accuracy of READ COMMITTED when defining transactions that involve ECP.

For further details, refer to Transaction Processing.

ISOLATION LEVEL in Effect

You can set the ISOLATION LEVEL for a process using **SET TRANSACTION** (without starting a transaction), **START TRANSACTION** (setting isolation mode and starting a transaction), or a **SetOption("IsolationMode")** method call.

The specified ISOLATION LEVEL remains in effect until explicitly reset by a **SET TRANSACTION**, **START TRANSACTION**, or a **SetOption("IsolationMode")** method call. Because **COMMIT** or **ROLLBACK** is only meaningful for changes to the data, not data queries, a **COMMIT** or **ROLLBACK** operation has no effect on the ISOLATION LEVEL setting.

The ISOLATION LEVEL in effect at the start of a query remains in effect for the duration of the query.

You can determine the ISOLATION LEVEL for the current process using the **GetOption("IsolationMode")** method call. You can also set the isolation mode for the current process using the **SetOption("IsolationMode")** method call. These methods specify READ UNCOMMITTED (the default) as 0, READ COMMITTED as 1, and READ VERIFIED as 3. Specifying any other numeric value leaves the isolation mode unchanged. No error or change occurs if you set the isolation mode to the current isolation mode. Use of these methods is shown in the following example:

ObjectScript

```
WRITE $SYSTEM.SQL.Util.GetOption("IsolationMode")," default",!
&sql(START TRANSACTION ISOLATION LEVEL READ COMMITTED,READ WRITE)
WRITE $SYSTEM.SQL.Util.GetOption("IsolationMode")," after START TRANSACTION",!
DO $SYSTEM.SQL.Util.SetOption("IsolationMode",0,.stat)
IF stat=1 {
  WRITE $SYSTEM.SQL.Util.GetOption("IsolationMode")," after IsolationMode=0 call",! }
ELSE { WRITE "Set IsolationMode error" }
&sql(COMMIT)
```

The isolation mode and the access mode must always be compatible. Changing the access mode changes the isolation mode, as shown in the following example:

ObjectScript

```
WRITE $SYSTEM.SQL.Util.GetOption("IsolationMode")," default",!
&sql(SET TRANSACTION ISOLATION LEVEL READ COMMITTED,READ WRITE)
WRITE $SYSTEM.SQL.Util.GetOption("IsolationMode")," after SET TRANSACTION",!
&sql(START TRANSACTION READ ONLY)
WRITE $SYSTEM.SQL.Util.GetOption("IsolationMode")," after changing access mode",!
&sql(COMMIT)
```

Arguments

commitmode

An optional argument specifying how future transactions will be committed to the database during the current process. Valid values are EXPLICIT, IMPLICIT, and NONE. The default is to maintain the existing commit mode; the initial commit mode default for a process is IMPLICIT.

transactionmodes

An optional argument that specifies the isolation mode and access mode for the transaction. You can specify a value for either an isolation mode, an access mode, or for both modes as a comma-separated list.

Valid values for isolation mode are ISOLATION LEVEL READ COMMITTED, ISOLATION LEVEL READ UNCOMMITTED, and ISOLATION LEVEL READ VERIFIED. The default is ISOLATION LEVEL READ UNCOMMITTED.

Valid values for access mode are READ ONLY and READ WRITE. Note that only ISOLATION LEVEL READ COMMITTED is compatible with access mode READ WRITE.

ObjectScript and SQL Transactions

ObjectScript and SQL transaction commands are fully compatible and interchangeable, with the following exception:

ObjectScript **TSTART** and SQL **START TRANSACTION** both start a transaction if no transaction is current. However, **START TRANSACTION** does not support nested transactions. Therefore, if you need (or may need) nested transactions, it is preferable to start the transaction with **TSTART**. If you need compatibility with the SQL standard, use **START TRANSACTION**.

ObjectScript transaction processing provides limited support for nested transactions. SQL transaction processing supplies support for savepoints within transactions.

If a transaction involves SQL data modification statements, the transaction should be started with the SQL **START TRANSACTION** statement and committed with the SQL **COMMIT** statement. (These statements may be explicit or implicit, depending on the %COMMITMODE setting.) Methods that use **TSTART**/**TCOMMIT** nesting can be included in the transaction, as long as they don't initiate the transaction. Methods and stored procedures should not normally use SQL transaction control statements, unless, by design, they are the main controller of the transaction. Stored procedures should not normally use SQL transaction control statements, because these stored procedures are normally called from ODBC/JDBC, which has its own model of transaction control.

Examples

The following Embedded SQL example uses two **START TRANSACTION** statements to start a transaction and set its parameters. Note that the first **START TRANSACTION** initiates a transaction, setting the commit mode and incrementing the **$TLEVEL** transaction level counter. The second **START TRANSACTION** sets the isolation mode for query read operations in the current transaction, but does not increment **$TLEVEL**, because the transaction has already been started. The **SAVEPOINT** statement increments **$TLEVEL**:

ObjectScript

```
   WRITE !,"Transaction level=",$TLEVEL
 &sql(START TRANSACTION %COMMITMODE EXPLICIT)
   WRITE !,"Start transaction commit mode, SQLCODE=",SQLCODE
   WRITE !,"Transaction level=",$TLEVEL
 &sql(START TRANSACTION ISOLATION LEVEL READ COMMITTED)
   WRITE !,"Start transaction isolation mode, SQLCODE=",SQLCODE
   WRITE !,"Transaction level=",$TLEVEL
 &sql(SAVEPOINT a)
   WRITE !,"Set Savepoint a, SQLCODE=",SQLCODE
   WRITE !,"Transaction level=",$TLEVEL
 &sql(COMMIT)
   WRITE !,"Commit transaction, SQLCODE=",SQLCODE
   WRITE !,"Transaction level=",$TLEVEL
```

See Also

*   COMMIT, %INTRANSACTION, ROLLBACK, SAVEPOINT, SET TRANSACTION, $TLEVEL

*   Transaction Processing

TRAIN MODEL (SQL)

Trains a machine learning model.

Synopsis

```
TRAIN MODEL model-name
      [ AS preferred-name ]
      [ NOT DEFAULT]
      [ FOR label-column ]
      [ WITH feature-column-clause ]
      [ FROM model-source ]
      [ USING json-object ]
```

Arguments

model-name	The name of the machine learning model to train.
AS *preferred-name*	*Optional* — An alternative name to save the trained model as. See details below.
NOT DEFAULT	*Optional* — A clause to train a model without setting it as the default trained model. See details below.
FOR *label-column*	*Optional* — The name of the column being predicted, aka, the label column. See details below.
WITH *feature-column-clause*	*Optional* — Inputs to the model, aka the feature columns, as either the name of a column or as a comma-separated list of the names of columns.
FROM *model-source*	The table or view from which the model is being built. This can be a table, view, or results of a join. See details below.
USING *json-object-string*	*Optional* — A JSON string specifying one or more key-value pairs. See details below.

Description

The **TRAIN MODEL** statement tells a provider to train a model using the specified model definition. The provider is specified by the ML configuration.

FROM

The **FROM** clause supplies the data for training your model.

- This clause is *required* if your **CREATE MODEL** statement did NOT specify a **FROM** clause.

- This clause is *optional* if your **CREATE MODEL** statement specified a **FROM** clause.

Examples highlighting acceptable use and omission of **FROM**:

FROM in TRAIN MODEL

```
CREATE MODEL model_b PREDICTING ( label ) WITH ( column_1, column_2, column_3)
TRAIN MODEL model_b FROM table
```

FROM in CREATE MODEL

```
CREATE MODEL model_a PREDICTING ( label ) FROM table
TRAIN MODEL model_a
```

Note: Omitting **FROM** from your **TRAIN MODEL** statement means that you use the default query from **CREATE MODEL**.

WITH

WITH allows you to explicitly match the feature columns in your data to the model definition schema. Each column is a standard identifier.

FOR

FOR allows you to explicitly match the label column in your data to the model definition schema. For example, if your label column in your model definition is named column_a but is named column_b in your training data, you can match the columns as follows:

```
CREATE MODEL model_a PREDICTING ( column_a ) FROM table_a
TRAIN MODEL model_a FOR column_b FROM table_b
```

Naming

AS allows you to explicitly name your trained model.

Model definitions and trained models exist in the same schema. If a trained model is not explicitly named with **AS**, its name consists of the model definition name with an appended running integer. We can see the difference by querying the INFORMATION_SCHEMA.ML_TRAINED_MODELS table:

```
CREATE MODEL TitanicModel PREDICTING (Survived binary) FROM IntegratedML_dataset_titanic.passenger
TRAIN MODEL TitanicModel
TRAIN MODEL TitanicModel
TRAIN MODEL TitanicModel
TRAIN MODEL TitanicModel AS TrainedTitanic
SELECT MODEL_NAME, TRAINED_MODEL_NAME FROM INFORMATION_SCHEMA.ML_TRAINED_MODELS
```

MODEL_NAME	TRAINED_MODEL_NAME
TitanicModel	TitanicModel_t1
TitanicModel	TitanicModel_t2
TitanicModel	TitanicModel_t3
TitanicModel	TrainedTitanic

Not Default

Each model definition has a default trained model. Without user-specification, the most recently trained model becomes the default. Using the **NOT DEFAULT** clause allows you to train a new model without the result becoming the default trained model:

```
CREATE MODEL TitanicModel PREDICTING (Survived) FROM IntegratedML_dataset_titanic.passenger
TRAIN MODEL TitanicModel As FirstModel
TRAIN MODEL TitanicModel As SecondModel NOT DEFAULT
SELECT MODEL_NAME, DEFAULT_TRAINED_MODEL_NAME FROM INFORMATION_SCHEMA.ML_MODELS
```

MODEL_NAME	DEFAULT_TRAINED_MODEL_NAME
TitanicModel	FirstModel

Without using **NOT DEFAULT**, the DEFAULT_TRAINED_MODEL field would otherwise read "SecondModel"

USING Clause Considerations

You can pass provider-specific parameters in a **USING** clause for a more customized training run. This clause accepts a JSON string with one or more key-value pairs. The list of parameters that you can use depends on the provider.

For instance, when training with AutoML as your provider you can change the random seed:

```
TRAIN MODEL IsSpam USING {"seed": 3}
```

See Providers for information about which parameters you can pass for each provider.

Passing NULL Values

Passing data with NULL values in the label column, in a **TRAIN MODEL** statement, will result in a trained model with undefined behavior. Users should carefully screen for NULL values as part of their data preparation process.

Required Security Privileges

Calling **TRAIN MODEL** requires %MANAGE_MODEL privileges; otherwise, there is a SQLCODE –99 error (Privilege Violation). To assign %MANAGE_MODEL privileges, use the GRANT command.

Examples

```
TRAIN MODEL EmailFilter

TRAIN MODEL model_5 AS MyModel USING {"seed": 3}

TRAIN MODEL LoanDefault FROM LoanData
```

See Also

- CREATE MODEL, VALIDATE MODEL

TRUNCATE TABLE (SQL)

Removes all data from a table and resets counters.

Synopsis

```
TRUNCATE TABLE [restriction] tablename
```

Description

The **TRUNCATE TABLE** command removes all rows from a table and resets all table counters. You can truncate a table directly, or through a view. Truncating a table through a view is subject to delete requirements and restrictions, as described in CREATE VIEW.

TRUNCATE TABLE resets the internal counters used for generating RowID field, IDENTITY field, and SERIAL (%Library.Counter) field sequential integer values. InterSystems IRIS assigns a value of 1 for these fields in the first row inserted into a table following a **TRUNCATE TABLE**. Performing a DELETE on all rows of a table does not reset these internal counters.

TRUNCATE TABLE resets the internal counter used for generating stream field OID values when data is inserted into a stream field. Performing a DELETE on all rows of a table does not reset this internal counter.

TRUNCATE TABLE always sets the %ROWCOUNT local variable to `-1`; it does not set %ROWCOUNT to the number of rows deleted.

TRUNCATE TABLE does not reset the ROWVERSION counter.

TRUNCATE TABLE suppresses the pulling of base table triggers that are otherwise pulled during **DELETE** processing. Because **TRUNCATE TABLE** performs a delete with %NOTRIGGER behavior, the user must have been granted the %NOTRIGGER privilege (using the GRANT statement) in order to run **TRUNCATE TABLE**. This aspect of **TRUNCATE TABLE** is functionally identical to:

SQL

```
DELETE %NOTRIGGER FROM tablename
```

Note: The DELETE command can also be used to delete all rows from a table. **DELETE** provides more functionality than **TRUNCATE TABLE**, including returning the number of rows deleted in %ROWCOUNT. **DELETE** does not reset internal counters.

TRUNCATE TABLE provides compatibility for code migration from other database software.

To truncate a table:

- The table must exist in the current (or specified) namespace. If the specified table cannot be located, InterSystems IRIS issues an SQLCODE -30 error.

- The user must have the %NOTRIGGER administrative privilege, even if no triggers are defined. Failing to have this privilege results in an SQLCODE –99 error with the %msg `User does not have %NOTRIGGER privileges`.

- The user must have DELETE privilege for the table. Failing to have this privilege results in an SQLCODE -99 with the %msg `User 'name' is not privileged for the operation`. You can determine if the current user has DELETE privilege by invoking the %CHECKPRIV command. You can determine if a specified user has DELETE privilege by invoking the **$SYSTEM.SQL.Security.CheckPrivilege()** method. For privilege assignment, refer to the GRANT command.

- The table cannot be defined as READONLY. Attempting to compile a **TRUNCATE TABLE** that references a read-only table results in an SQLCODE -115 error. Note that this error is now issued at compile time, rather than only occurring at execution time. See the description of READONLY objects in Other Options for Persistent Classes.

- If deleting through a view, the view must be updateable; it cannot be defined as WITH READ ONLY. Attempting to do so results in an SQLCODE -35 error. See CREATE VIEW for further details.

- All of the rows must be available for deletion. By default, if one or more rows cannot be deleted, the **TRUNCATE TABLE** operation fails and no rows are deleted.

   TRUNCATE TABLE fails if the table is locked by another process in either EXCLUSIVE MODE or SHARE MODE. Attempting a **TRUNCATE TABLE** operation on a locked table results in an SQLCODE -110 error, with a %msg such as the following: `Unable to acquire lock for DELETE of table 'Sample.MyStuff' on row with RowID = '3'` (where the specified RowID is the first row in the table).

   TRUNCATE TABLE fails if deleting a row would violate foreign key referential integrity. No rows are deleted and **TRUNCATE TABLE** issues an SQLCODE -124 error. This default behavior is modifiable, as described below.

Atomicity

TRUNCATE TABLE does not occur within an automatically initiated transaction, and therefore no journaling or rollback option is provided.

If journaling and the option to rollback **TRUNCATE TABLE** is required, you must explicitly specify a **START TRANSACTION** and conclude with an explicit **COMMIT** or **ROLLBACK**.

This is the same as SET TRANSACTION %COMMITMODE= NONE or 0 (no auto transaction) — No transaction is initiated when you invoke **TRUNCATE TABLE**. A failed **TRUNCATE TABLE** operation can leave the database in an inconsistent state, with some rows deleted and some not deleted. To provide transaction support in this mode you must use **START TRANSACTION** to initiate the transaction and **COMMIT** or **ROLLBACK** to end the transaction.

TRUNCATE TABLE for a sharded table is always performed using SET TRANSACTION %COMMITMODE NONE, even when the user has explicitly set SET TRANSACTION %COMMITMODE EXPLICIT.

Restriction Arguments

To use a *restriction* argument, you must have the corresponding *admin-privilege* for the current namespace. Refer to GRANT for further details.

Specifying *restriction* argument(s) restricts processing as follows:

- %NOCHECK — suppress referential integrity checking for foreign keys that reference the rows being deleted.

- %NOLOCK — suppress row locking of the rows being deleted. This should only be used when a single user/process is updating the database.

- %NOJOURN — suppress journaling of the rows being deleted and disable transactions for the duration of the deletions. None of the changes made in any of the rows are journaled, including any triggers fired. If you perform a **ROLLBACK** after a statement with %NOJOURN, the changes made by the statement will not be rolled back.

You can specify multiple *restriction* arguments in any order. Multiple arguments are separated by spaces.

If you specify a *restriction* argument when deleting a parent record, the same *restriction* argument will be applied when deleting the corresponding child records.

TRUNCATE TABLE always performs a delete with implicit %NOTRIGGER behavior, and requires the corresponding *admin-privilege*.

Referential Integrity

InterSystems IRIS uses the system-wide configuration setting to determine whether to perform foreign key referential integrity checking; the default is to perform foreign key referential integrity checking. You can set this default system-wide, as described in Foreign Key Referential Integrity Checking. To determine the current system-wide setting, call **$SYSTEM.SQL.CurrentSettings()**.

During a **TRUNCATE TABLE** operation, for every foreign key reference, a shared lock is acquired on the corresponding row in the referenced table. This row is locked until the end of the transaction. This ensures that the referenced row is not changed before a potential rollback of the **TRUNCATE TABLE**.

Transaction Locking

InterSystems IRIS performs standard locking on a **TRUNCATE TABLE** operation. Unique field values are locked for the duration of the current transaction.

The default lock threshold is 1000 locks per table, which means if you delete more than 1000 unique field values from a table during a transaction, the lock threshold is reached and InterSystems IRIS automatically elevates the locking level from unique field value locks to a table lock. This permits large-scale deletes during a transaction without overflowing the lock table.

You can determine the current system-wide lock threshold value using the **$SYSTEM.SQL.Util.GetOption("LockThreshold")** method. This system-wide lock threshold value is configurable:

- Using the **$SYSTEM.SQL.Util.SetOption("LockThreshold")** method.

- Using the Management Portal. Go to **System Administration**, **Configuration**, **SQL and Object Settings**, **SQL**. View and edit the current setting of **Lock escalation threshold**.

You must have USE permission on the %Admin Manage Resource to change the lock threshold. InterSystems IRIS immediately applies any change made to the lock threshold value to all current processes.

For further details on transaction locking refer to Transaction Processing.

Imported SQL Code

The **ImportDDL("IRIS")** and **Run()** methods do not support the **TRUNCATE TABLE** command. A **TRUNCATE TABLE** command found in an SQL code file imported by these methods is ignored. These import methods do support the **DELETE** command.

Arguments

restriction

An optional argument specifying one or more of the following restriction keywords, separated by spaces: %NOCHECK, %NOLOCK.

tablename

The table from which you are deleting all rows. You can also specify an updateable view through which you can delete all of the rows of a table. A table name (or view name) can be qualified (schema.table), or unqualified (table). An unqualified name is matched to its schema using either a schema search path (if provided) or the default schema name.

Examples

The following two Dynamic SQL examples compare **DELETE** and **TRUNCATE TABLE**. Each example creates a table, inserts rows into the table, deletes all the rows in the table, then inserts a single row into the now empty table.

The first example uses DELETE to delete all the records in the table. Note that **DELETE** does not reset the RowID counter:

ObjectScript

```
SET tcreate = "CREATE TABLE SQLUser.MyStudents (StudentName VARCHAR(32),StudentDOB DATE)"
SET tinsert = "INSERT INTO SQLUser.MyStudents (StudentName,StudentDOB) "_
              "SELECT Name,DOB FROM Sample.Person WHERE Age <= '21'"
SET tinsert1 = "INSERT INTO SQLUser.MyStudents (StudentName,StudentDOB) VALUES ('Bob Jones',60123)"
SET tdelete = "DELETE SQLUser.MyStudents"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(tcreate)
 IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
 WRITE rset.%StatementTypeName,!

 NEW %ROWCOUNT,%ROWID
 SET qStatus = tStatement.%Prepare(tinsert)
 IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
 WRITE rset.%StatementTypeName," rowcount ",rset.%ROWCOUNT,!

 SET qStatus = tStatement.%Prepare(tdelete)
 IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
 WRITE rset.%StatementTypeName," rowcount ",rset.%ROWCOUNT,!

SET qStatus = tStatement.%Prepare(tinsert1)
 IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
 WRITE rset.%StatementTypeName," rowcount ",rset.%ROWCOUNT," RowID ",rset.%ROWID,!
&sql(DROP TABLE SQLUser.MyStudents)
```

The second example uses **TRUNCATE TABLE** to delete all the records in the table. Note that *%StatementTypeName* returns "DELETE" for **TRUNCATE TABLE**. Note that **TRUNCATE TABLE** does reset the RowID counter:

ObjectScript

```
SET tcreate = "CREATE TABLE SQLUser.MyStudents (StudentName VARCHAR(32),StudentDOB DATE)"
SET tinsert = "INSERT INTO SQLUser.MyStudents (StudentName,StudentDOB) "_
              "SELECT Name,DOB FROM Sample.Person WHERE Age <= '21'"
SET tinsert1 = "INSERT INTO SQLUser.MyStudents (StudentName,StudentDOB) VALUES ('Bob Jones',60123)"
SET ttrunc = "TRUNCATE TABLE SQLUser.MyStudents"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(tcreate)
 IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
 WRITE rset.%StatementTypeName,!

NEW %ROWCOUNT,%ROWID
SET qStatus = tStatement.%Prepare(tinsert)
 IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
 WRITE rset.%StatementTypeName," rowcount ",rset.%ROWCOUNT,!

SET qStatus = tStatement.%Prepare(ttrunc)
 IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
 WRITE rset.%StatementTypeName," (TRUNCATE TABLE) rowcount ",rset.%ROWCOUNT,!

SET qStatus = tStatement.%Prepare(tinsert1)
 IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
 WRITE rset.%StatementTypeName," rowcount ",rset.%ROWCOUNT," RowID ",rset.%ROWID,!
&sql(DROP TABLE SQLUser.MyStudents)
```

See Also

- DELETE, INSERT, UPDATE

- CREATE VIEW

- Defining Tables

- Defining Views

- Transaction Processing

- SQL and Object Settings Pages

- SQLCODE error messages

TUNE TABLE (SQL)

Gathers table statistics based on representative data.

Synopsis

```
TUNE TABLE tablename [ tune_options ]
```

Description

The **TUNE TABLE** command gathers the statistics of an existing table based on the data currently in the table. This data should be representative of the data expected when the table is fully populated.

TUNE TABLE calculates and sets the BlockCount and extent size of the table, as well as the selectivity for each field, based on representative data. Normally, **TUNE TABLE** sets one or more of these values, and purges all cached queries that use this persistent class (table) so that queries will use these new values. However, if **TUNE TABLE** does not change any of these values (for example, if the data has not changed since the last time **TUNE TABLE** was run against this table) cached queries are not purged and the table's class definition is not flagged for recompile.

TUNE TABLE updates the SQL table definition (and therefore requires privileges to alter the table definition). Commonly, **TUNE TABLE** also updates the corresponding persistent class definition. This allows the gathered statistics to be used by the query optimizer without requiring a class compilation. However, if a class is deployed, **TUNE TABLE** only updates the SQL table definition; the query optimizer indirectly uses the gathered statistics from the table definition.

If **TUNE TABLE** is successful, it sets SQLCODE = 0. If the specified *tablename* does not exist, **TUNE TABLE** issues an SQLCODE -30 error.

Privileges

The **TUNE TABLE** command is a privileged operation. The user must have %ALTER_TABLE administrative privilege to execute **TUNE TABLE**. Failing to do so results in an SQLCODE –99 error with the %msg User 'name' does not have %ALTER_TABLE privileges. You can use the GRANT command to assign %ALTER_TABLE privileges to a user or role, if you hold appropriate granting privileges. Administrative privileges are namespace-specific. For further details, refer to Privileges.

The user must have %ALTER privilege on the specified table. If the user is the Owner (creator) of the table, the user is automatically granted %ALTER privilege for that table. Otherwise, the user must be granted %ALTER privilege for the table. Failing to do so results in an SQLCODE –99 error with the %msg User 'name' is not privileged for the operation. You can determine if the current user has %ALTER privilege by invoking the %CHECKPRIV command. You can use the GRANT command to assign %ALTER privilege to a specified table. For further details, refer to Privileges.

TUNE TABLE Options

- %CLEAR_VALUES: if specified the existing SELECTIVITY, EXTENTSIZE, etc. values are cleared from the class and table definition. Not specifying this option provides the default Tune Table behavior.

- %SAMPLE_PERCENT *percentage*: specifies the percentage of rows of the table to be used for sampling the data for the Tune Table utility. This *percentage* can be specified as '.##' or '##%'; for example, either '.12' or '12%' will cause the command to use 12% of the rows in the table when sampling the data. Specify *percentage* with a value greater than 0 and less than or equal to 100%; a value out of this range issues an SQLCODE -1 error.

    This value does not usually need to be specified. Only specify this value when potential outlier values for a field are not evenly distributed among rows throughout the table. Note, for any table with an extentsize < 1000 rows, the entire extent will be used by Tune Table regardless of the %SAMPLE_PERCENT value.

- %RECOMPILE_CQ: if specified, instead of just purging cached queries for the table that was tuned, Tune Table will instead recompile the cached query classes using the new Tune Table statistics. Not specifying this option provides the default Tune Table behavior.

If the specified *tune_options* value does not exist, **TUNE TABLE** issues an SQLCODE -25 error. If the same *tune_options* value is specified twice, **TUNE TABLE** issues an SQLCODE -326 error.

Cached Queries

Executing **TUNE TABLE** creates a cached query. The Show Plan display indicates that no Query Plan is created. No SQL Statement is created. The cached query is general to the namespace; it is not listed for the specific table. You can re-run the same **TUNE TABLE** statement using the cached query.

Executing **TUNE TABLE** purges all existing cached queries for the specified table, including the cached query for the previous execution of **TUNE TABLE**. You can optionally have **TUNE TABLE** recompile all of these cached queries with the new Tune Table values.

If running **TUNE TABLE** does not change any Tune Table values, cached queries are not purged.

Other Ways to Run Tune Table

There are two other interfaces for running Tune Table:

- Using the Management Portal SQL interface Actions drop-down list, which allows you to run Tune Table on a single table or on all of the tables in a schema.

- Invoking the **$SYSTEM.SQL.Stats.Table.GatherTableStats()** method for a single table, or all tables in the current namespace.

For further details, refer to Tune Table.

Arguments

tablename

The name of an existing table from which to gather statistics. The table name can be qualified (schema.table), or unqualified (table). An unqualified table name takes the default schema name.

tune_options

If specified, one or more TUNE TABLE options, specified in any order, separated by spaces. These *tune_options* are not case sensitive

Examples

The following example gathers table statistics and recompiles cached query classes based on the newly gathered statistics:

SQL

```
TUNE TABLE Sample.MyTest %RECOMPILE_CQ
```

The following example gathers table statistics from a 40% sample of the table:

SQL

```
TUNE TABLE Sample.MyTest %SAMPLE_PERCENT '40%'
```

See Also

- Tune Table

- ExtentSize, Selectivity, and BlockCount

UNFREEZE PLANS (SQL)

Unfreezes one or more frozen query plans.

Synopsis

```
UNFREEZE PLANS [[FROM] UPGRADE] BY ID statement-hash
UNFREEZE PLANS [[FROM] UPGRADE] BY TABLE table-name
UNFREEZE PLANS [[FROM] UPGRADE] BY SCHEMA schema-name
UNFREEZE PLANS [[FROM] UPGRADE]
```

Description

The **UNFREEZE PLANS** command unfreezes frozen query plans. To freeze query plans use the FREEZE PLANS command.

UNFREEZE PLANS without the FROM UPGRADE clause unfreezes all query plans with the Plan State `Frozen/Explicit`. **UNFREEZE PLANS** with the FROM UPGRADE clause unfreezes all query plans with the Plan State `Frozen/Upgrade`. The FROM keyword in this clause is optional.

UNFREEZE PLANS provides four syntax forms for unfreezing query plans:

- A specified query plan: **UNFREEZE PLANS BY ID** *statement-hash*. The *statement-hash* value must be delimited by double quotation marks.

- All query plans for a table: **UNFREEZE PLANS BY TABLE** *table-name*. You can specify a table name or a view name. If a query plan references multiple tables and/or views, specifying any of these tables or views unfreezes the query plan.

- All query plans for all tables in a schema: **UNFREEZE PLANS BY SCHEMA** *schema-name*.

- All query plans for all tables in the current namespace: **UNFREEZE PLANS**.

This command issues SQLCODE 0 if one or more query plans are unfrozen; it issues SQLCODE 100 if no query plans are unfrozen. The Rows Affected (%ROWCOUNT) indicates the number of query plans unfrozen.

Other Interfaces

You can use the following $SYSTEM.SQL.Statement methods to unfreeze a single query plan or multiple query plans: **UnfreezeStatement()** for a single plan; **UnfreezeRelation()** for all plans for a relation (a table or view referenced in the query plan); **UnfreezeSchema()** for all plans for a schema; **UnfreezeAll()** for all plans in the current namespace. There are corresponding Freeze methods.

You can use the Management Portal, to unfreeze a query plan, as described in Frozen Plans Interface.

Arguments

statement-hash

The internal hash representation of the SQL Statement definition for a query plan, enclosed in quotation marks. Occasionally, what appear to be identical SQL statements may have different statement hash entries. Any difference in settings/options that require different code generation of the SQL statement result in a different statement hash. This may occur with different client versions or different platforms that support different internal optimizations. Refer to SQL Statement Details.

table-name

The name of an existing table or view. A *table-name* can be qualified (schema.table), or unqualified (table). An unqualified table name takes the default schema name.

schema-name

The name of an existing schema. This command unfreezes all frozen query plans for all tables in the specified schema.

Security and Privileges

The **UNFREEZE PLANS** command is a privileged operation that required the user to have %Development:USE permission. Such permissions can be granted through the Management Portal. Executing a **UNFREEZE PLANS** command without this privileges will result in a SQLCODE -99 error and the command will fail. There are two exceptions:

- The command is executed via Embedded SQL, which does not perform privilege checks.

- The user explicitly specifies not privilege checking by, for example, calling either **%Prepare()** with the checkPriv argument set to 0 or **%ExecDirectNoPriv()** on a %SQL.Statement.

See Also

- FREEZE PLANS command

- Frozen Plans

- SQL Statements

UNLOCK (SQL)

Unlocks a table.

Synopsis

```
UNLOCK [TABLE] tablename IN EXCLUSIVE MODE [IMMEDIATE]
UNLOCK [TABLE] tablename IN SHARE MODE [IMMEDIATE]
```

Description

The **UNLOCK** command unlocks an SQL table that was locked by the **LOCK** command. This table must be an existing table for which you have the necessary privileges. If *tablename* is a temporary table, the command completes successfully, but performs no operation. If *tablename* is a view, the command fails with an SQLCODE -400 error.

UNLOCK and **UNLOCK TABLE** are synonymous.

The **UNLOCK** command reverses the **LOCK** operation. The **UNLOCK** command completes successfully even when no lock is held. You can use **LOCK** to lock a table multiple times; you must explicitly **UNLOCK** the table as many times as it was explicitly locked.

Privileges

The **UNLOCK** command is a privileged operation. Prior to using **UNLOCK** IN SHARE MODE it is necessary for your process to have SELECT privilege for the specified table. Prior to using **UNLOCK** IN EXCLUSIVE MODE it is necessary for your process to have INSERT, UPDATE, or DELETE privilege for the specified table. For IN EXCLUSIVE MODE, the INSERT or UPDATE privilege must be on at least one field of the table. Failing to hold sufficient privileges results in an SQLCODE -99 error (Privilege Violation). You can determine if the current user has the necessary privileges by invoking the %CHECKPRIV command. You can determine if a specified user has the necessary table-level privileges by invoking the **$SYSTEM.SQL.Security.CheckPrivilege()** method. For privilege assignment, refer to the GRANT command.

Nonexistent Table

If you try to unlock a nonexistent table, **UNLOCK** fails with a compile error, and the message `SQLCODE=-30 : Table 'SQLUser.mytable' not found`.

Arguments

tablename

The name of the table to be unlocked. *tablename* must be an existing table. A *tablename* can be qualified (schema.table), or unqualified (table). An unqualified table name takes the default schema name. A schema search path is ignored.

IN EXCLUSIVE MODE / IN SHARE MODE

The IN EXCLUSIVE MODE keyword phrase releases a regular InterSystems IRIS lock. The IN SHARE MODE keyword phrase releases a shared lock at the InterSystems IRIS level.

IMMEDIATE

An optional argument. If not specified, InterSystems IRIS releases the lock at the end of the current transaction. If specified, InterSystems IRIS releases the lock immediately.

Examples

The following embedded SQL examples create a table, lock it and then unlock it:

ObjectScript

```
NEW SQLCODE,%msg
&sql(CREATE TABLE mytest (
     ID NUMBER(12,0) NOT NULL,
     CREATE_DATE DATE DEFAULT CURRENT_TIMESTAMP(2),
     WORK_START DATE DEFAULT SYSDATE) )
IF SQLCODE=0 { WRITE !,"Table created" }
ELSE { WRITE !,"CREATE TABLE error: ",SQLCODE
       QUIT }
```

ObjectScript

```
NEW SQLCODE,%msg
&sql(LOCK mytest IN EXCLUSIVE MODE)
IF SQLCODE=0 { WRITE !,"Table locked" }
ELSEIF SQLCODE=-110 { WRITE !,"Table is locked by another process",!,%msg }
ELSE { WRITE !,"Unexpected LOCK error: ",SQLCODE,!,%msg }
&sql(UNLOCK mytest IN EXCLUSIVE MODE)
IF SQLCODE=0 { WRITE !,"Table unlocked" }
ELSE { WRITE !,"Unexpected UNLOCK error: ",SQLCODE,!,%msg }
```

See Also

- LOCK

- INSERT UPDATE DELETE

- SQLCODE error messages

UPDATE (SQL)

Sets new values for specified columns in a specified table.

Synopsis

```
UPDATE [%keyword] table-ref [[AS] t-alias]
    SET column1 = scalar-expression1 {,column2 = scalar-expression2} ...
    [FROM [optimize-option] select-table [[AS] t-alias] {, select-table2 [[AS] t-alias]} ]
    [WHERE condition-expression]
UPDATE [%keyword] table-ref [[AS] t-alias] [ (column1 {,column2} ...) ]
    VALUES (scalar-expression1 {,scalar-expression2} ...)
    [FROM [optimize-option] select-table [[AS] t-alias] {, select-table2 [[AS] t-alias]} ]
    [WHERE condition-expression]
UPDATE [%keyword] table-ref [[AS] t-alias]
    VALUES :array()
    [FROM [optimize-option] select-table [[AS] t-alias] {, select-table2 [[AS] t-alias]} ]
    [WHERE condition-expression]

UPDATE [%keyword] table-ref [[AS] t-alias]
    SET column1 = scalar-expression1{,column2 = scalar-expression2} ...
    [WHERE CURRENT OF cursor]
UPDATE [%keyword] table-ref [[AS] t-alias] [ (column1 {,column2} ...) ]
    VALUES (scalar-expression1 {,scalar-expression2} ...)
    [WHERE CURRENT OF cursor]
UPDATE [%keyword] table-ref [[AS] t-alias]
    VALUES :array()
    [WHERE CURRENT OF cursor]
```

Description

An **UPDATE** command changes existing values for columns in a table. You can update data in a table directly, update through a view, or update using a subquery enclosed in parentheses. Updating through a view is subject to requirements and restrictions, as described in CREATE VIEW.

The **UPDATE** command provides one or more new column values to one or more existing base table rows that contain those columns. Assignment of data values to columns is done using a *value-assignment-statement*. By default, a *value-assignment-statement* updates all rows in the table.

More commonly, an **UPDATE** specifies the updating of a specific row (or rows) based on a *condition-expression*. By default, an **UPDATE** operation goes through all of the rows of a table and updates all rows that satisfy the *condition-expression*. If no rows satisfy the *condition-expression*, **UPDATE** completes successfully and sets SQLCODE=100 (No more data).

You can specify a **WHERE** clause or a **WHERE CURRENT OF** clause (but not both). If the **WHERE CURRENT OF** clause is used, **UPDATE** updates the record at the current position of the cursor. For details on positioned operations, see WHERE CURRENT OF.

The **UPDATE** operation sets the %ROWCOUNT local variable to the number of updated rows, and the %ROWID local variable to the RowID value of the last row updated.

By default, the **UPDATE** operation is an all-or-nothing event. Either all specified rows and columns are updated, or none are.

For more details regarding the different ways of assigning values to columns, refer to the Value Assignment section below.

INSERT OR UPDATE

The INSERT OR UPDATE statement is a variant of the **INSERT** statement that performs both insert and update operations. First it attempts to perform an insert operation. If the insert request fails due to a UNIQUE KEY violation (for the field(s) of some unique key, there exists a row that already has the same value(s) as the row specified for the insert), then it automatically turns into an update request for that row, and **INSERT OR UPDATE** uses the specified field values to update the existing row.

SQLCODE Errors

By default, a multi-row **UPDATE** is an atomic operation. If one or more rows cannot be updated, the **UPDATE** operation fails and no rows are updated. InterSystems IRIS sets the SQLCODE variable, which indicates the success or failure of the **UPDATE**; if the operation failed, IRIS also sets %msg.

To update a table, the update must meet all table, column name, and value requirements, as follows.

Tables:

- The table must exist in the current (or specified) namespace. If the specified table cannot be located, InterSystems IRIS issues an SQLCODE -30 error.

- The table cannot be defined as *READONLY*. Attempting to compile an **UPDATE** that references a read-only table results in an SQLCODE -115 error. Note that this error is issued at compile time, rather than occurring at execution time. See the description of READONLY objects in Other Options for Persistent Classes.

- The table cannot be locked IN EXCLUSIVE MODE by another process. Attempting to update a locked table results in an SQLCODE -110 error, with a %msg such as the following: `Unable to acquire lock for UPDATE of table 'Sample.Person' on row with RowID = '10'`. Note that an SQLCODE -110 error occurs only when the **UPDATE** statement locates the first record to be updated, then cannot lock it within the timeout period.

- If the **UPDATE** specifies a non-existent field, an SQLCODE -29 is issued. To list all of the field names defined for a specified table, refer to Column Names and Numbers. If the field exists but none of the field values fulfill the **UPDATE** command's WHERE clause, no rows are affected and SQLCODE 100 (end of data) is issued.

- In rare cases, **UPDATE** with %NOLOCK locates a row to be updated, but then the row is immediately deleted by another process; this situation results in an SQLCODE -109 error: `Cannot find the row designated for UPDATE`. The %msg for this error lists the table name and the RowID.

- If updating a table through a view, the view cannot be defined as WITH READ ONLY. Attempting to do so results in an SQLCODE -35 error. If the view is based on a sharded table, you cannot **UPDATE** through a view defined WITH CHECK OPTION. Attempting to do so results in an SQLCODE -35 with the %msg `INSERT/UPDATE/DELETE not allowed for view (sample.myview) based on sharded table with check option conditions`. See CREATE VIEW for further details.

Column Names and Values:

- The update cannot include duplicate field names. Attempting an update that specifies two fields with the same name results in an SQLCODE -377 error.

- You cannot update a field that has been locked by another concurrent process. Attempting to do so results in an SQLCODE -110 error. This SQLCODE error can also occur if you are performing such a large number of updates that a <LOCKTABLEFULL> error occurs.

- You cannot update integer counter fields. These fields are non-modifiable. Attempting to do so generates the following errors: RowID field (SQLCODE -107); IDENTITY field (SQLCODE -107); SERIAL (%Library.Counter) field (SQLCODE -105); ROWVERSION field (SQLCODE -138). The field values for these fields are system-generated and not user-modifiable. Even when the user can insert an initial value for a counter field, the user cannot update the value.

  The one exception is when adding a SERIAL (%Library.Counter) field to a table that has existing data. Existing records will have NULL for this added counter field. In this case, you can use **UPDATE** to change a NULL to an integer value. See ALTER TABLE for further details.

- You cannot update a shard key field. Attempting a update a field that is part of a shard key generates an SQLCODE -154 error.

- You cannot update a field value if the update would violate the field's uniqueness constraints. Attempting to update the value of a field (or group of fields) such that the update would violate a uniqueness constraint or a primary key constraint results in an SQLCODE -120 error. This error is returned if the field has a UNIQUE data constraint, or if

the unique fields constraint has been applied to a group of fields. The SQLCODE -120 %msg string includes both the field and the value that violate the uniqueness constraint. For example `<Table 'Sample.MyTable', Constraint 'MYTABLE_UNIQUE3', Field(s) FullName="Molly Bloom"; failed unique check>` or `<Table 'Sample.MyTable', Constraint 'MYTABLE_PKEY2', Field(s) FullName="Molly Bloom"; failed unique check>`. For details on listing a table's unique value and primary key field constraints and the naming of constraints, refer to Catalog Details: Constraints.

- You cannot update a field value if the update specifies a value that is not listed in its VALUELIST parameter. A property of a persistent class defined with a VALUELIST parameter can only accept as a valid value one of the values listed in VALUELIST, or be provided with no value (NULL). VALUELIST valid values are case-sensitive. Attempting to update with a data value that doesn't match the VALUELIST values results in an SQLCODE -105 field value failed validation error.

- Numbers are inserted in canonical form, but can be specified with leading and trailing zeros and multiple leading signs. However, in SQL, two consecutive minus signs are parsed as a single-line comment indicator. Therefore, attempting to specify a number with two consecutive leading minus signs results in an SQLCODE -12 error.

- When using a **WHERE CURRENT OF** clause, you cannot update a field using the current field value to generate an updated value. For example, `SET Salary=Salary+100` or `SET Name=UPPER(Name)`. Attempting to do so results in an SQLCODE -69 error: SET <field> = <value expression> not allowed with WHERE CURRENT OF <cursor>.

- If updating one of the specified rows would violate foreign key referential integrity (and %NOCHECK is not specified), the **UPDATE** fails to update any rows and instead issues an SQLCODE -124 error. This does not apply if the foreign key was defined with the NOCHECK keyword.

- You cannot update a non-stream field with stream data. This results in an SQLCODE -303 error, as described below.

- Inserted data values must pass display to logical mode conversion. InterSystems SQL stores data in *logical* mode format. For some data types, the logical format might differ from the display format. For example, date data is stored as an integer count of days, time data is stored as a count of seconds from midnight, and %List data is stored as an encoded string. Other data types, such as strings and numbers, require no conversion. Attempting to insert a value in a format that cannot be converted to its logical storage value results in an error (SQLCODE -146 for dates, SQLCODE -147 for times). For more details on mode conversions, see Data Display Options.

Arguments

%keyword

An optional argument specifying one or more of the following keyword options, separated by spaces: %NOCHECK, %NOFPLAN, %NOINDEX, %NOJOURN, %NOLOCK, %NOTRIGGER, %PROFILE, %PROFILE_ALL.

table-ref

The name of an existing table where data is to be updated. You can also specify a view through which to perform the update on a table. You cannot specify a table-valued function or JOIN syntax in this argument.

A table name (or view name) can be qualified (schema.table), or unqualified (table). An unqualified name is matched to its schema using either a schema search path (if provided) or the default schema name.

AS t-alias

An optional alias for a *table-ref* (table or view) name. An alias must be a valid identifier. The AS keyword is optional

FROM select-table

An optional FROM clause used to specify the table or tables used to determine which rows are to be updated.

Multiple tables can be specified as a comma-separated list or associated with ANSI join keywords. Any combination of tables or views can be specified. If you specify a comma between two *select-table*s here, InterSystems IRIS performs a

CROSS JOIN on the tables and retrieves data from the results table of the JOIN operation. If you specify ANSI join keywords between two *select-table*s here, InterSystems IRIS performs the specified join operation. For further details, refer to JOIN.

You can optionally specify one or more *optimize-option* keywords to optimize query execution. The available options are: %ALLINDEX, %FIRSTTABLE *select-table*, %FULL, %INORDER, %IGNOREINDICES, %NOFLATTEN, %NOMERGE, %NOSVSO, %NOTOPOPT, %NOUNIONOROPT, %PARALLEL, and %STARTTABLE. See FROM for further details.

WHERE condition-expression

An optional argument that specifies one or more boolean predicates used to determine which rows are to be updated. If a **WHERE** clause (or a **WHERE CURRENT OF** clause) is not supplied, **UPDATE** updates all the rows in the table. See WHERE for further details.

WHERE CURRENT OF cursor

An optional argument specifying that the **UPDATE** operation updates the record at the current position of cursor. You can specify a **WHERE CURRENT OF** clause or a **WHERE** clause, but not both. For further details, see WHERE CURRENT OF.

column

An optional argument specifying the name of an existing column. Multiple column names are specified as a comma-separated list. If omitted, all columns are updated.

scalar-expression

A column data value expressed as a scalar expression. Multiple data values are specified as a comma-separated list with each data value corresponding in sequence to a column.

:array()

Embedded SQL only — An array of values specified as a host variable. The lowest subscript level of the array must be unspecified. Thus `:myupdates()`, `:myupdates(5,)`, and `:myupdates(1,1,)` are all valid specifications.

Value Assignment

You can assign new values to specified columns in a variety of ways.

- Using the SET keyword, specify one or more column = scalar-expression pairs as a comma-separated list. For example:

  ```
  SET StatusDate='05/12/06',Status='Purged'
  ```

- Using the VALUES keyword, specify a list of columns equated to a corresponding scalar-expressions list. For example:

  ```
  (StatusDate,Status) VALUES ('05/12/06','Purged')
  ```

  When assigning scalar-expression values to a column list, there must be a scalar-expression for each specified column.

- Using the VALUES keyword without a column list, specify a list of scalar-expressions that implicitly correspond to the columns of the row in column order. The following example specifies all of the columns in the table, specifying a literal value to update the Address column:

  ```
  VALUES (Name,DOB,'22 Main St. Anytown MA 12345',SSN)
  ```

  When assigning values to an implicit column list, you must supply a value for every updateable field, in the order that the columns are defined in the DDL. (You do not specify the non-updateable RowID column.) These values can either be a literal to specify a new value, or the field name to specify the existing value. You cannot specify placeholder commas or omit trailing fields.

- Using the VALUES keyword without a column list, specify a subscripted array in which the numeric subscripts correspond to the column numbers, including in your column count the non-updateable RowID as column number 1. For example:

```
VALUES :myarray()
```

This value assignment can only be performed from Embedded SQL using a host variable. Unlike all other value assignments, this usage allows you to delay specifying which columns are to be updated until runtime (by populating the array at runtime). All other types of update require that the columns to be updated must be specified at compile time. This syntax cannot be used with a linked table; attempting to do so results in an SQLCODE=-155 error. For further details, see Host Variable as a Subscripted Array.

For program examples demonstrating each of these types of **UPDATE**, refer to the Examples section below.

List Structures

InterSystems IRIS supports the list structure data type %List (data type class %Library.List). This is a compressed binary format, which does not map to a corresponding native data type for InterSystems SQL. It corresponds to data type VARBINARY with a default MAXLEN of 32749. For this reason, Dynamic SQL cannot use **UPDATE** or **INSERT** to set a property value of type %List. For further details, refer to Data Types.

Stream Values

You can update data values in a stream field as follows:

- For any table: A string literal or a host variable containing a string literal, for example:

ObjectScript

```
SET literal="update stream string value"
    //do the update; use a string
&sql(UPDATE MyStreamTable SET MyStreamField = :literal WHERE %ID=21)
```

- For a non-sharded table: An object reference (OREF) to a stream object. InterSystems IRIS opens this object and copies its contents, updating the stream field. For example:

ObjectScript

```
SET oref=##class(%Stream.GlobalCharacter).%New()
DO oref.Write("Update stream string value non-shard 1")
    //do the update; use an actual OREF
&sql(UPDATE MyStreamTable SET MyStreamField = :oref WHERE %ID=22)
```

or a string version of an OREF of a stream, for example:

ObjectScript

```
SET oref=##class(%Stream.GlobalCharacter).%New()
DO oref.Write("Update stream string value non-shard 2")
    //next line converts OREF to a string OREF
set string=oref_""
    //do the update
&sql(UPDATE MyStreamTable SET MyStreamField = :string WHERE %ID=23)
```

- For a sharded table: An object ID (OID) using a temporary stream object stored in the ^IRIS.Stream.Shard global:

ObjectScript

```
SET clob=##class(%Stream.GlobalCharacter).%New("Shard")
DO clob.Write("Update sharded table stream string value")
SET sc=clob.%Save() // Handle $$$ISERR(sc)
set ClobOid=clob.%Oid()
//do the update
&sql(UPDATE MyStreamTable SET MyStreamField = :ClobOid WHERE %ID=24)
```

You cannot update a non-Stream field with the contents of a Stream field. This results in an SQLCODE -303 error: "No implicit conversion of Stream value to non-Stream field in UPDATE assignment is supported". To update a string field with Stream data, you must first use the **SUBSTRING** function to convert the first *n* characters of the Stream data to a string, as shown in the following example:

SQL

```
UPDATE MyTable
    SET MyStringField=SUBSTRING(MyStreamField,1,2000)
```

Computed Fields

A field defined with COMPUTECODE may recompute its value as part of the UPDATE operation, as follows:

- COMPUTECODE: value is computed and stored upon INSERT, value is not changed upon UPDATE.

- COMPUTECODE with COMPUTEONCHANGE: value is computed and stored upon INSERT, is recomputed and stored upon UPDATE.

- COMPUTECODE with DEFAULT and COMPUTEONCHANGE: default value is stored upon INSERT, value is computed and stored upon UPDATE. If the compute code contains a programming error (for example, divide by zero), the UPDATE operation fails with an SQLCODE -415 error.

- COMPUTECODE with CALCULATED or TRANSIENT: you cannot UPDATE a value for this field because no value is stored. The value is computed when queried. However, if you attempt to update a value in a calculated field, InterSystems IRIS performs validation on the supplied value and issues an error if the value is invalid. If the value is valid, InterSystems IRIS performs no update operation, issues no SQLCODE error, and increments ROWCOUNT.

A COMPUTEONCHANGE computed field is not recomputed when no actual update occurs: when the UPDATE operation new field value is the same as the prior field value.

In most cases, you define a computed field as read-only. This prevents an update operation directly changing a value that is intended to be the result of a computation involving other field values. In this case, attempting to use **UPDATE** to overwrite the value of a computed field results in an SQLCODE -138 error.

However, you may wish to revise a computed field value to reflect an update to one (or more) of its source field values. You can do this by using an update trigger that recomputes the computed field value after you have updated a specified source field. For example, an update to the Salary data field might trip a trigger that recalculates the Bonus computed field. This update trigger recalculates Bonus and completes successfully, even when Bonus is a read-only field. See CREATE TRIGGER.

You can use the **CREATE TABLE** ON UPDATE keyword phrase to define a field that is set to a literal or a system variable (such as the current timestamp) when the record is updated.

For further details, refer to Computing a field value on INSERT or UPDATE.

%SerialObject Properties

When updating data in a %SerialObject, you must update the table (persistent class) that references the embedded %SerialObject; you cannot update a %SerialObject directly. From the referencing table, you can either:

- Use the referencing field to update values for multiple %SerialObject properties as a %List structure. For example, if the persistent class has a property PAddress that references a serial object contain the properties Street, City, and Country (in that order), you update SET PAddress=$LISTBUILD('123 Main St.','Newtown','USA') or (PAddress) VALUES ($LISTBUILD('123 Main St.','Newtown','USA')) or (PAddress) VALUES (:vallist). The %List must contain values for the properties of the serial object (or placeholder commas) in the order that these properties are specified in the serial object.

This type of update may not perform validation of %SerialObject property values. Therefore, it is strongly suggested that you use the **$SYSTEM.SQL.Schema.ValidateTable()** method to perform Table Data Validation after updating %SerialObject property values using a %List structure.

- Use underscore syntax to update values for individual %SerialObject properties in any order. For example, if the persistent class has a property PAddress that references a serial object contain the properties Street, City, and Country, you update `SET PAddress_City='Newtown',PAddress_Street='123 Main St.',PAddress_Country='USA'`.

  This type of update performs validation of %SerialObject property values.

FROM Clause

An **UPDATE** command may have no FROM keyword. It may simply specify the table (or view) to update, and select which rows to update using a WHERE clause.

However, you can also include an optional FROM clause after the *value-assignment-statement*. This FROM clause specifies one or more tables used to determine which records are to be updated. The FROM clause is commonly, but not always, used with a WHERE clause involving multiple tables. A FROM clause can be complex, and can include ANSI join syntax. Any syntax supported in a **SELECT** FROM clause is permitted in an **UPDATE** FROM clause. This **UPDATE** FROM clause provides functionality compatibility with Transact-SQL.

The following example shows how this FROM clauses might be used. It updates those records from the Employees table where the same EmpId is also found in the Retirees table:

SQL

```
UPDATE Employees AS Emp
    SET retired='Yes'
    FROM Retirees AS Rt
    WHERE Emp.EmpId = Rt.EmpId
```

If the **UPDATE** *table-ref* and the FROM clause make reference to the same table, these references may either be to the same table, or to a join of two instances of the table. This depends on how table aliases are used:

- If neither table reference has an alias, both reference the same table:

  ```
  UPDATE table1 value-assignment FROM table1,table2   /* join of 2 tables */
  ```

- If both table references have the same alias, both reference the same table:

  ```
  UPDATE table1 AS x value-assignment FROM table1 AS x,table2   /* join of 2 tables */
  ```

- If both table references have aliases, and the aliases are different, InterSystems IRIS performs a join of two instances of the table:

  ```
  UPDATE table1 AS x value-assignment FROM table1 AS y,table2   /* join of 3 tables */
  ```

- If the first table reference has an alias, and the second does not, InterSystems IRIS performs a join of two instances of the table:

  ```
  UPDATE table1 AS x value-assignment FROM table1,table2   /* join of 3 tables */
  ```

- If the first table reference does not have an alias, and the second has a single reference to the table with an alias, both reference the same table, and this table has the specified alias:

  ```
  UPDATE table1 value-assignment FROM table1 AS x,table2   /* join of 2 tables */
  ```

- If the first table reference does not have an alias, and the second has more than one reference to the table, InterSystems IRIS considers each aliased instance a separate table and performs a join on these tables:

```
UPDATE table1 value-assignment FROM table1,table1 AS x,table2      /* join of 3 tables */
UPDATE table1 value-assignment FROM table1 AS x,table1 AS y,table2  /* join of 4 tables */
```

%Keyword Arguments

Specifying *%keyword* argument(s) restricts processing as follows:

- %NOCHECK — Unique value checking and foreign key referential integrity checking are not performed. Column data validation for data type, maximum length, data constraints, and other validation criteria is also not performed. The WITH CHECK OPTION validation for a view is not performed when performing an **UPDATE** through a view.

  Note: Because use of %NOCHECK can result in invalid data, this *%keyword* argument should only be used when performing bulk inserts or updates from a reliable data source.

  The user must have the corresponding %NOCHECK administrative privilege for the current namespace to apply this restriction. Failing to do so results in an SQLCODE –99 error with the %msg User 'name' does not have %NOCHECK privileges.

  If you wish to prevent updates that result in non-unique data values when specifying %NOCHECK, perform an EXISTS check prior to **UPDATE**.

  If you wish to disable only foreign key referential integrity checking, use the **$SYSTEM.SQL.Util.SetOption("FilerRefIntegrity")** method rather than specifying %NOCHECK. Alternatively, a foreign key can be defined with the NOCHECK keyword, so that foreign key referential integrity checking is never performed.

- %NOFPLAN — FROM clause syntax only: the frozen plan (if any) is ignored for this operation; the operation generates a new query plan. The frozen plan is retained, but not used. For further details, refer to Frozen Plans.

- %NOINDEX — the index maps are not set during **UPDATE** processing. The user must have the corresponding %NOINDEX administrative privilege for the current namespace to apply this restriction. Failing to do so results in an SQLCODE -99 error.

- %NOJOURN — suppress journaling and disable transactions for the duration of the update operation. None of the changes made in any of the rows are journaled, including any triggers pulled. However, updates are still journaled in a mirrored environment. If you perform a **ROLLBACK** after a statement with %NOJOURN, the changes made by the statement will not be rolled back. The user must have the corresponding %NOJOURN administrative privilege for the current namespace to apply this restriction. Failing to do so results in an SQLCODE -99 error

- %NOLOCK — the row is not locked upon **UPDATE**. This should only be used when a single user/process is updating the database. The user must have the corresponding %NOLOCK administrative privilege for the current namespace to apply this restriction. Failing to do so results in an SQLCODE -99 error.

- %NOTRIGGER — the base table triggers are not pulled during **UPDATE** processing. Neither BEFORE nor AFTER triggers are executed. The user must have the corresponding %NOTRIGGER administrative privilege for the current namespace to apply this restriction. Failing to do so results in an SQLCODE -99 error.

- %PROFILE or %PROFILE_ALL — if one of these keyword directives is specified, SQLStats collecting code is generated. This is the same code that would be generated with PTools turned ON. The difference is that SQLStats collecting code is only generated for this specific statement. All other SQL statements within the routine/class being compiled will generate code as if PTools is turned OFF. This enables the user to profile/inspect specific problem SQL statements within an application without collecting irrelevant statistics for SQL statements that are not being investigated. For further details, refer to SQL Runtime Statistics.

%PROFILE collects SQLStats for the main query module. %PROFILE_ALL collects SQLStats for the main query module and all of its subquery modules.

You can specify multiple *%keyword* arguments in any order. Multiple arguments are separated by spaces.

Referential Integrity

If you do not specify %NOCHECK, InterSystems IRIS uses the system-wide configuration setting to determine whether to perform foreign key referential integrity checking; the default is to perform foreign key referential integrity checking. You can set this default system-wide, as described in Foreign Key Referential Integrity Checking. To determine the current system-wide setting, call **$SYSTEM.SQL.CurrentSettings()**.

This setting does not apply to foreign keys that have been defined with the NOCHECK keyword.

During an **UPDATE** operation, for every foreign key reference which has a field value being updated, a shared lock is acquired on both the old (pre-update) referenced row and the new (post-update) referenced row in the referenced table(s). These rows are locked while performing referential integrity checking and updating the row. The lock is then released (it is not held until the end of the transaction). This ensures that the referenced row is not changed between the referential integrity check and the completion of the update operation. Locking the old row ensures that the referenced row is not changed before a potential rollback of the **UPDATE**. Locking the new row ensures that the referenced row is not changed between the referential integrity checking and the completion of the update operation.

If an **UPDATE** operation with %NOLOCK is performed on a foreign key field defined with CASCADE, SET NULL, or SET DEFAULT, the corresponding referential action changing the foreign key table is also performed with %NOLOCK.

Atomicity

By default, **UPDATE**, **INSERT**, **DELETE**, and **TRUNCATE TABLE** are atomic operations. An **UPDATE** either completes successfully or the whole operation is rolled back. If any of the specified rows cannot be updated, none of the specified rows are updated and the database reverts to its state before issuing the **UPDATE**.

You can modify this default for the current process within SQL by invoking SET TRANSACTION %COMMITMODE. You can modify this default for the current process in ObjectScript by invoking the **SetOption()** method, as follows `SET status=$SYSTEM.SQL.Util.SetOption("AutoCommit",intval,.oldval)`. The following *intval* integer options are available:

- 1 or IMPLICIT (autocommit on) — The default behavior, as described above. Each **UPDATE** constitutes a separate transaction.

- 2 or EXPLICIT (autocommit off) — If no transaction is in progress, an **UPDATE** automatically initiates a transaction, but you must explicitly **COMMIT** or **ROLLBACK** to end the transaction. In EXPLICIT mode the number of database operations per transaction is user-defined.

- 0 or NONE (no auto transaction) — No transaction is initiated when you invoke **UPDATE**. A failed **UPDATE** operation can leave the database in an inconsistent state, with some of the specified rows updated and some not updated. To provide transaction support in this mode you must use **START TRANSACTION** to initiate the transaction and **COMMIT** or **ROLLBACK** to end the transaction.

  A sharded table is always in no auto transaction mode, which means all inserts, updates, and deletes to sharded tables are performed outside the scope of a transaction.

You can determine the atomicity setting for the current process using the **GetOption("AutoCommit")** method, as shown in the following ObjectScript example:

ObjectScript

```
SET stat=$SYSTEM.SQL.Util.SetOption("AutoCommit",$RANDOM(3),.oldval)
IF stat'=1 {WRITE "SetOption failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET x=$SYSTEM.SQL.Util.GetOption("AutoCommit")
  IF x=1 {
    WRITE "Default atomicity behavior",!
    WRITE "automatic commit or rollback" }
  ELSEIF x=0 {
    WRITE "No transaction initiated, no atomicity:",!
    WRITE "failed DELETE can leave database inconsistent",!
    WRITE "rollback is not supported" }
  ELSE { WRITE "Explicit commit or rollback required" }
```

Transaction Locking

If you do not specify %NOLOCK, the system automatically performs standard record locking on **INSERT**, **UPDATE**, and **DELETE** operations. Each affected record (row) is locked for the duration of the current transaction.

The default lock threshold is 1000 locks per table. This means that if you update more than 1000 records from a table during a transaction, the lock threshold is reached and InterSystems IRIS automatically escalates the locking level from record locks to a table lock. This permits large-scale updates during a transaction without overflowing the lock table.

InterSystems IRIS applies one of the two following lock escalation strategies:

- "E"-type lock escalation: InterSystems IRIS uses this type of lock escalation if the following are true: (1) the class uses %Storage.Persistent (you can determine this from the Catalog Details in the Management Portal SQL schema display). (2) the class either does not specify an IDKey index, or specifies a single-property IDKey index. "E"-type lock escalation is described in LOCK.

- Traditional SQL lock escalation: The most likely reason why a class would not use "E"-type lock escalation is the presence of a multi-property IDKey index. In this case, each %Save increments the lock counter. This means if you do 1001 saves of a single object within a transaction, InterSystems IRIS will attempt to escalate the lock.

For both lock escalation strategies, you can determine the current system-wide lock threshold value using the **$SYSTEM.SQL.Util.GetOption("LockThreshold")** method. The default is 1000. This system-wide lock threshold value is configurable:

- Using the **$SYSTEM.SQL.Util.SetOption("LockThreshold")** method.

- Using the Management Portal. Go to **System Administration**, **Configuration**, **SQL and Object Settings**, **SQL**. View and edit the current setting of **Lock escalation threshold**. The default is 1000 locks. If you change this setting, any new process started after changing it will have the new setting.

You must have USE permission on the %Admin Manage Resource to change the lock threshold. InterSystems IRIS immediately applies any change made to the lock threshold value to all current processes.

One potential consequence of automatic lock escalation is a deadlock situation that might occur when an attempt to escalate to a table lock conflicts with another process holding a record lock in that table. There are several possible strategies to avoid this: (1) increase the lock escalation threshold so that lock escalation is unlikely to occur within a transaction. (2) substantially lower the lock escalation threshold so that lock escalation occurs almost immediately, thus decreasing the opportunity for other processes to lock a record in the same table. (3) apply a table lock for the duration of the transaction and do not perform record locks. This can be done at the start of the transaction by specifying LOCK TABLE, then UNLOCK TABLE (without the IMMEDIATE keyword, so that the table lock persists until the end of the transaction), then perform updates with the %NOLOCK option.

Automatic lock escalation is intended to prevent overflow of the lock table. However, if you perform such a large number of updates that a <LOCKTABLEFULL> error occurs, **UPDATE** issues an SQLCODE -110 error.

For further details on transaction locking refer to Transaction Processing.

Counter Incrementing

ROWVERSION Counter Increment

If a table has a field of data type ROWVERSION, performing an update on a row automatically updates the integer value of this field. The ROWVERSION field takes the next sequential integer from the namespace-wide row version counter. Attempting to specify an update value to a ROWVERSION field results in an SQLCODE -138 error.

SERIAL (%Counter) Counter Increment

An **UPDATE** operation has no effect on SERIAL (%Library.Counter) counter field values. However, an update performed using INSERT OR UPDATE causes a skip in integer sequence for subsequent insert operations for a SERIAL field. Refer to INSERT OR UPDATE for further details.

Privileges

To perform an update, you must either have table-level UPDATE privilege for the specified table (or view) or column-level UPDATE privilege for the specified column(s). When updating all fields in a row, note that column-level privileges cover all table columns named in the **GRANT** command; table-level privileges cover all table columns, including those added after the privilege was assigned.

- The user must have UPDATE privilege on the specified table, or column-level UPDATE privilege for all columns in the update field list.

- The user must have SELECT privilege for fields in a WHERE clause, whether or not those fields are to be updated. You must have both SELECT and UPDATE privileges for those fields if they are included in the update field list. In the following example, the Name field must have (at least) column-level SELECT privilege:

SQL

```
UPDATE Sample.Employee (Salary) VALUES (1000000) WHERE Name='Smith, John'
```

In the above example, the Salary field requires only column-level UPDATE privilege.

If the user is the Owner (creator) of the table, the user is automatically granted all privileges for that table. Otherwise, the user must be granted privileges for the table. Failing to do so results in an SQLCODE –99 error with the %msg `User 'name' is not privileged for the operation`. You can determine if the current user has the appropriate privileges by invoking the %CHECKPRIV command. You can use the GRANT command to assign the user table privileges. For further details, refer to Privileges.

When a property is defined as ReadOnly, the corresponding table field is also defined as ReadOnly. A ReadOnly field may only be assigned a value using InitialExpression or SqlComputed. Attempting to update a value (even a NULL value) for a field for which you have column-level ReadOnly (SELECT or REFERENCES) privilege results in an SQLCODE -138 error: `Cannot INSERT/UPDATE a value for a read only field`. When you link a table using the Link Table Wizard, you have the option of defining fields as Read Only. The field on the source system might not be read only, but if InterSystems IRIS defines the linked table's field as Read Only, attempting an **UPDATE** that references this field results in an SQLCODE -138 error.

Row-Level Security

InterSystems IRIS row-level security permits **UPDATE** to modify any row that security permits it to access. It allows you to update a row even if the update creates a row that security will not permit you to subsequently access. To ensure that an update does not prevent you from subsequent **SELECT** access to the row, it is recommended that you perform the **UPDATE** through a view that has a WITH CHECK OPTION. For further details, refer to CREATE VIEW.

Examples

The examples in this section update the SQLUser.MyStudents table. The following example creates the SQLUser.MyStudents table and populates it with data. Because repeated execution of this example would accumulate records with duplicate data, it uses **TRUNCATE TABLE** to remove old data before invoking **INSERT**. Execute this example before invoking the **UPDATE** examples:

ObjectScript

```
CreateStudentTable
    SET stuDDL=5
    SET stuDDL(1)="CREATE TABLE SQLUser.MyStudents ("
    SET stuDDL(2)="StudentName VARCHAR(32),StudentDOB DATE,"
    SET stuDDL(3)="StudentAge INTEGER COMPUTECODE {SET {StudentAge}="
    SET stuDDL(4)="$PIECE(($PIECE($H,"","",1)-{StudentDOB})/365,"".""."",1)} CALCULATED,"
    SET stuDDL(5)="Q1Grade CHAR,Q2Grade CHAR,Q3Grade CHAR,FinalGrade VARCHAR(2))"
  SET tStatement = ##class(%SQL.Statement).%New(0,"Sample")
  SET qStatus = tStatement.%Prepare(.stuDDL)
    IF qStatus'=1 {WRITE "DDL %Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET rtn = tStatement.%Execute()
  IF rtn.%SQLCODE=0 {WRITE !,"Table Create successful"}
  ELSEIF rtn.%SQLCODE=-201 {WRITE "Table already exists, SQLCODE=",rtn.%SQLCODE,!}
  ELSE {WRITE !,"table create failed, SQLCODE=",rtn.%SQLCODE,!
        WRITE rtn.%Message,! }
RemoveOldData
  SET clearit="TRUNCATE TABLE SQLUser.MyStudents"
  SET qStatus = tStatement.%Prepare(clearit)
   IF qStatus'=1 {WRITE "Truncate %Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET truncrtn = tStatement.%Execute()
  IF truncrtn.%SQLCODE=0 {WRITE !,"Table old data removed",!}
  ELSEIF truncrtn.%SQLCODE=100 {WRITE !,"no data to be removed",!}
  ELSE {WRITE !,"truncate failed, SQLCODE=",truncrtn.%SQLCODE," ",truncrtn.%Message,! }
PopulateStudentTable
  SET studentpop=2
  SET studentpop(1)="INSERT INTO SQLUser.MyStudents (StudentName,StudentDOB) "
  SET studentpop(2)="SELECT Name,DOB FROM Sample.Person WHERE Age <= '21'"
  SET qStatus = tStatement.%Prepare(.studentpop)
    IF qStatus'=1 {WRITE "Populate %Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET poprtn = tStatement.%Execute()
  IF poprtn.%SQLCODE=0 {WRITE !,"Table Populate successful",!
        WRITE poprtn.%ROWCOUNT," rows inserted"}
  ELSE {WRITE !,"table populate failed, SQLCODE=",poprtn.%SQLCODE,!
        WRITE poprtn.%Message }
```

You can use the following query to display the results of these examples:

SQL

```
SELECT %ID,* FROM SQLUser.MyStudents ORDER BY StudentAge,%ID
```

Some of the following **UPDATE** examples depend on field values set by other **UPDATE** examples; they should be run in the order specified.

In the following Dynamic SQL example, a `SET field=value` **UPDATE** modifies a specified field in selected records. In the MyStudents table, children under the age of 7 are not given grades:

ObjectScript

```
    SET studentupdate=3
    SET studentupdate(1)="UPDATE SQLUser.MyStudents "
    SET studentupdate(2)="SET FinalGrade='NA' "
    SET studentupdate(3)="WHERE StudentAge <= 6"
  SET tStatement = ##class(%SQL.Statement).%New(0,"Sample")
  SET qStatus = tStatement.%Prepare(.studentupdate)
    IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET uprtn = tStatement.%Execute()
  IF uprtn.%SQLCODE=0 {WRITE !,"Table Update successful"
                       WRITE !,"Rows updated=",uprtn.%ROWCOUNT," Final RowID=",uprtn.%ROWID}
  ELSE {WRITE !,"Table update failed, SQLCODE=",uprtn.%SQLCODE," ",uprtn.%Message }
```

In the following cursor-based Embedded SQL example, a `SET field1=value1,field2=value2` **UPDATE** modifies several fields in selected records. In the MyStudents table, it updates specified student records with Q1 and Q2 grades:

ObjectScript

```
#sqlcompile path=Sample
NEW %ROWCOUNT,%ROWID
&sql(DECLARE StuCursor CURSOR FOR
     SELECT * FROM MyStudents
     WHERE %ID IN(10,12,14,16,18,20,22,24) AND StudentAge > 6)
 &sql(OPEN StuCursor)
     QUIT:(SQLCODE'=0)
 FOR { &sql(FETCH StuCursor)
     QUIT:SQLCODE
     &sql(Update MyStudents SET Q1Grade='A',Q2Grade='A'
     WHERE CURRENT OF StuCursor)
  IF SQLCODE=0 {
  WRITE !,"Table Update successful"
  WRITE !,"Row count=",%ROWCOUNT," RowID=",%ROWID }
  ELSE {
  WRITE !,"Table Update failed, SQLCODE=",SQLCODE }
  }
  &sql(CLOSE StuCursor)
```

In the following Dynamic SQL example, a `field-list VALUES value-list` **UPDATE** modifies the values of several fields in selected records. In the MyStudents table, children who don't receive a final grade also don't receive quarterly grades:

ObjectScript

```
   SET studentupdate=3
   SET studentupdate(1)="UPDATE SQLUser.MyStudents "
   SET studentupdate(2)="(Q1Grade,Q2Grade,Q3Grade) VALUES ('x','x','x') "
   SET studentupdate(3)="WHERE FinalGrade='NA'"
 SET tStatement = ##class(%SQL.Statement).%New()
 SET qStatus = tStatement.%Prepare(.studentupdate)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
 SET uprtn = tStatement.%Execute()
 IF uprtn.%SQLCODE=0 {WRITE !,"Table Update successful"
                     WRITE !,"Rows updated=",uprtn.%ROWCOUNT," Final RowID=",uprtn.%ROWID}
 ELSE {WRITE !,"Table Update failed, SQLCODE=",uprtn.%SQLCODE," ",uprtn.%Message,! }
```

In the following Dynamic SQL example, a `VALUES value-list` **UPDATE** modifies all the field values in selected records. Note that this syntax requires that you specify a value for every field in the record. In the MyStudents table, several children have been withdrawn from school. Their record IDs and names are retained, with the word WITHDRAWN appended to the name; all other data is removed and the DOB field is used for the withdrawal date:

ObjectScript

```
   SET studentupdate=4
   SET studentupdate(1)="UPDATE SQLUser.MyStudents "
   SET studentupdate(2)="VALUES (StudentName||' WITHDRAWN',"
   SET studentupdate(3)="$PIECE($HOROLOG,',',1),00,'-','-','-','XX') "
   SET studentupdate(4)="WHERE %ID IN(7,10,22)"
 SET tStatement = ##class(%SQL.Statement).%New()
 SET qStatus = tStatement.%Prepare(.studentupdate)
   IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
 SET uprtn = tStatement.%Execute()
 IF uprtn.%SQLCODE=0 {WRITE !,"Table Update successful"
                     WRITE !,"Rows updated=",uprtn.%ROWCOUNT," Final RowID=",uprtn.%ROWID}
 ELSE {WRITE !,"Table Update failed, SQLCODE=",uprtn.%SQLCODE," ",uprtn.%Message,! }
```

In the following Dynamic SQL example, a `subquery` **UPDATE** uses a subquery to select records. It then modifies these records using `SET field=value` syntax. Because of the way that StudentAge is calculated from date of birth in SQLUser.MyStudents, anyone less than a year old has a calculated age of <Null>, and anyone whose date of birth has been nulled has a very high calculated age. Here the StudentName field is flagged for future confirmation of the date of birth:

ObjectScript

```
   SET studentupdate=3
   SET studentupdate(1)="UPDATE (SELECT StudentName FROM SQLUser.MyStudents "
   SET studentupdate(2)="WHERE StudentAge IS NULL OR StudentAge > 21) "
   SET studentupdate(3)="SET StudentName = StudentName||' *** CHECK DOB' "
 SET tStatement = ##class(%SQL.Statement).%New(0,"Sample")
 SET qStatus = tStatement.%Prepare(.studentupdate)
   IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
 SET uprtn = tStatement.%Execute()
 IF uprtn.%SQLCODE=0 {WRITE !,"Table Update successful"
                     WRITE !,"Rows updated=",uprtn.%ROWCOUNT," Final RowID=",uprtn.%ROWID}
 ELSE {WRITE !,"Table Update failed, SQLCODE=",uprtn.%SQLCODE," ",uprtn.%Message,! }
```

In the following Embedded SQL example, a `VALUES :array()` **UPDATE** modifies the field values specified by column number in the array in selected records. A `VALUES :array()` update can only be done in Embedded SQL. Note that this syntax requires that you specify each value by DDL column number (including in your column count the RowID column (column 1) but supplying no value to this non-modifiable field). In the MyStudents table, children between 4 and 6 (inclusive) are given a 'P' (for 'Present') in their Q1Grade (column 5) and Q2Grade (column 6) fields. All other record data remains unchanged:

ObjectScript

```
 SET arry(5)="P"
 SET arry(6)="P"
 &sql(UPDATE SQLUser.MyStudents VALUES :arry()
     WHERE FinalGrade='NA' AND StudentAge > 3)
 IF SQLCODE=0 {WRITE "Table Update successful",!
               WRITE "Rows updated=",%ROWCOUNT," Final RowID=",%ROWID }
 ELSE {WRITE "Table Update failed, SQLCODE=",SQLCODE,! }
```

See Also

- INSERT

- INSERT OR UPDATE

- DELETE

- SELECT

- VALUES

- FROM

- WHERE

- WHERE CURRENT OF

- CREATE TABLE

- CREATE VIEW

- Modifying the Database

- Defining Tables

- Defining Views

- Transaction Processing

- SQL and Object Settings Pages

- SQLCODE error messages

USE DATABASE (SQL)

Sets the current namespace and database.

Synopsis

```
USE [DATABASE] dbname
```

Description

The **USE DATABASE** command switches the current process to the specified namespace and its associated database. This allows you to change namespaces within SQL. The DATABASE keyword is optional.

The specified *dbname* is the name of the desired namespace and corresponding directory that contains the database files. Specify *dbname* as an identifier. Namespace names are not case-sensitive. For further information on using namespaces, see Namespaces and Databases.

Because USER is an SQL Reserved Word, you must use a delimited identifier to specify the USER namespace, as shown in the following SQL Shell example:

```
USER>>USE DATABASE Samples
SAMPLES>>USE DATABASE "User"
USER>>
```

If the specified *dbname* does not exist, InterSystems IRIS issues an SQLCODE -400 error.

The **USE DATABASE** command is a privileged operation. Prior to using **USE DATABASE**, it is necessary to be logged in as a user with appropriate privileges. Failing to do so results in an SQLCODE -99 error (Privilege Violation).

Use the **$SYSTEM.Security.Login()** method to assign a user with appropriate privileges:

ObjectScript

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql(        )
```

You must have the **%Service_Login:Use** privilege to invoke the **$SYSTEM.Security.Login** method. For further information, see %SYSTEM.Security.

You can also switch to a different namespace using the ObjectScript ZNSPACE command, or the SET $NAMESPACE statement.

Executing via a Database Driver

When the **USE DATABASE** command is executed via a database driver, the server process performs a simulated connection reset. Data structures used by the server process are cleaned up. However, commit mode is not changed. The Read Committed setting is not changed either. If a transaction is in process, the transaction simply continues and is not committed or rolled back.

Arguments

dbname

The namespace and corresponding database to be used by the current process as the current namespace.

See Also

- CREATE DATABASE command

- DROP DATABASE command

VALIDATE MODEL (SQL)

Validates a model.

Synopsis

```
VALIDATE MODEL model-name [ AS validation-run-name ]
       [ USE trained-model-name]
       [ WITH feature-column-clause ]
       FROM model-source
```

Arguments

model-name	The name of a model to validate.
AS *validation-run-name*	*Optional* — A name to save your validation run as. See details below.
USE *trained-model-name*	*Optional* — The name of a non-default trained model to be validated. See details below.
WITH *feature-column-clause*	*Optional* — The specific columns from your dataset that you want to use for validating your model.
FROM *model-source*	The table or view from which the model is being validated. This can be a table, view, or results of a join. See details below.

Description

The **VALIDATE MODEL** command calculates validation metrics for a given trained model, based on its performance on a specified testing dataset. Each command creates a *validation run*.

Naming

AS allows you to explicitly name your validation run.

If a validation run is not explicitly named with AS, its name consists of the trained model with an appended running integer. We can see the difference by querying the INFORMATION_SCHEMA.ML_VALIDATION_RUNS table:

```
CREATE MODEL TitanicModel PREDICTING (Survived) FROM IntegratedML_dataset_titanic.passenger
TRAIN MODEL TitanicModel
VALIDATE MODEL TitanicModel FROM IntegratedML_dataset_titanic.passenger
VALIDATE MODEL TitanicModel FROM IntegratedML_dataset_titanic.passenger
VALIDATE MODEL TitanicModel FROM IntegratedML_dataset_titanic.passenger
VALIDATE MODEL TitanicModel AS TitanicValidation FROM IntegratedML_dataset_titanic.passenger
SELECT MODEL_NAME, TRAINED_MODEL_NAME, VALIDATION_RUN_NAME FROM INFORMATION_SCHEMA.ML_VALIDATION_RUNS
```

MODEL_NAME	TRAINED_MODEL_NAME	VALIDATION_RUN_NAME
TitanicModel	TitanicModel_t1	TitanicModel_t1_v1
TitanicModel	TitanicModel_t1	TitanicModel_t1_v2
TitanicModel	TitanicModel_t1	TitanicModel_t1_v3
TitanicModel	TitanicModel_t1	TitanicValidation

USE

USE allows you to specify the trained model to perform validation on. If a trained model is not explicitly named by **USE**, the statement validates the default trained model for the specified model definition.

We can see the difference by querying the INFORMATION_SCHEMA.ML_VALIDATION_RUNS table:

```
CREATE MODEL TitanicModel PREDICTING (Survived) FROM IntegratedML_dataset_titanic.passenger
TRAIN MODEL TitanicModel AS FirstModel
TRAIN MODEL TitanicModel AS SecondModel
TRAIN MODEL TitanicModel AS ThirdModel
VALIDATE MODEL TitanicModel FROM IntegratedML_dataset_titanic.passenger
VALIDATE MODEL TitanicModel FROM IntegratedML_dataset_titanic.passenger
VALIDATE MODEL TitanicModel USE FirstModel FROM IntegratedML_dataset_titanic.passenger
VALIDATE MODEL TitanicModel USE SecondModel FROM IntegratedML_dataset_titanic.passenger
SELECT MODEL_NAME, TRAINED_MODEL_NAME FROM INFORMATION_SCHEMA.ML_VALIDATION_RUNS
```

MODEL_NAME	TRAINED_MODEL_NAME
TitanicModel	ThirdModel
TitanicModel	ThirdModel
TitanicModel	FirstModel
TitanicModel	SecondModel

FROM Considerations

While you used a *training* set to train your model, you should use other data, a *testing* data set, to validate your model. Using your training data to validate a model only evaluates goodness of fit, as opposed to evaluating the model's predictive performance on other data.

This data should be of the same schema as your training data, including the feature columns and label column.

Required Security Privileges

Calling **VALIDATE MODEL** requires %USE_MODEL privileges; otherwise, there is a SQLCODE –99 error (Privilege Violation). To assign %USE_MODEL privileges, use the GRANT command.

Validation Metrics

The output of **VALIDATE MODEL** is a set of validation metrics that is viewable in the INFORMATION_SCHEMA.ML_VALIDATION_METRICS table.

For regression models, the following metrics are saved:

- Variance

- R-squared

- Mean squared error

- Root mean squared error

For classification models, the following metrics are saved:

- Precision — This is calculated by dividing the number of true positives by the number of predicted positives (sum of true positives and false positives).

- Recall — This is calculated by dividing the number of true positives by the number of actual positives (sum of true positives and false negatives).

- F-Measure — This is calculated by the following expression:

  ```
  F = 2 * (precision * recall) / (precision + recall)
  ```

- Accuracy — This is calculated by dividing the number of true positives and true negatives by the total number of rows (sum of true positives, false positives, true negatives, and false negatives) across the entire test set.

Examples

```
VALIDATE MODEL PatientReadmission FROM Patient_test
VALIDATE MODEL PatientReadmission AS PatientValidation USE PatientReadmission_H2OModel FROM Patient_test
```

See Also

- CREATE MODEL, TRAIN MODEL, PREDICT

SQL Clauses

DISTINCT (SQL)

A SELECT clause that specifies to return only distinct values.

Synopsis

```
SELECT [DISTINCT [BY (item {,item2})] ]  |  [ALL]   select-item {,select-item2}
```

Arguments

Argument	Description
DISTINCT	*Optional* — Returns rows for which the combined *select-item* value(s) are unique.
DISTINCT BY (*item* {,*item2*})	*Optional* — Returns *select-item* values for rows for which the BY (*item*) value(s) are unique.
ALL	*Optional* — Returns all rows in the result set. The default.

Description

The optional **DISTINCT** clause appears after the **SELECT** keyword and before the optional TOP clause and the first *select-item*.

The **DISTINCT** clause is applied to the result set of the SELECT statement. It limits the rows returned to one arbitrary row for each distinct (unique) value. If no DISTINCT clause is specified, the default is to display all the rows that fulfill the **SELECT** criteria. The ALL clause is the same as specifying no DEFAULT clause; if you specify ALL, **SELECT** returns all the rows in the table that fulfill the **SELECT** criteria.

The **DISTINCT** clause has two forms:

- SELECT DISTINCT: Returns one row for each unique combination of *select-item* values. You can specify one or more than one *select-items*. For example, the following query returns a row with Home_State and Age values for each unique combination of Home_State and Age values:

  SQL

  ```
  SELECT DISTINCT Home_State,Age FROM Sample.Person
  ```

- SELECT DISTINCT BY (*item*): Returns one row for each unique combination of *item* values. You can specify a single *item* or a comma-separated list of *item*s. The specified *item* or *item* list must be enclosed in parentheses. Spaces may be specified or omitted between the BY keyword and the parentheses. The *select-item* list may, but does not have to, include the specified *item*(s). For example, the following query returns a row with Name and Age values for each unique combination of Home_State and Age values:

  SQL

  ```
  SELECT DISTINCT BY (Home_State,Age) Name,Age FROM Sample.Person
  ```

  The *item* field(s) must be specified by column name. Valid values include the following: a column name (`DISTINCT BY (City)`); an %ID (which returns all rows); a scalar function specifying a column name (`DISTINCT BY (ROUND(Age,-1))`); a collation function specifying a column name (`DISTINCT BY (%EXACT(City))`). You cannot specify a field by column alias; attempting to do so generates an SQLCODE -29 error. You cannot specify a field by column number; this is interpreted as a literal and returns one row. Specifying a literal as the *item* value in a **DISTINCT** clause returns 1 row; which row is returned is indeterminate. Thus, specifying 7, 'Chicago', '', 0, or NULL

all return 1 row. However, if you specify a literal as an *item* value in a comma-separated list, the literal is ignored and **DISTINCT** selects one arbitrary row for each unique combination of the specified field names.

The DISTINCT clause is applied before the TOP clause. If both are specified, the **SELECT** returns only rows with unique values, the number of unique value rows specified in the TOP clause.

If the column specified in the DISTINCT clause has rows that are NULL (contain no value), DISTINCT returns one row with NULL as a distinct (unique) value, as shown in the following examples:

SQL

```
SELECT DISTINCT FavoriteColors FROM Sample.Person
```

SQL

```
SELECT DISTINCT BY (FavoriteColors) Name,FavoriteColors FROM Sample.Person
ORDER BY FavoriteColors
```

A DISTINCT clause is not meaningful in an Embedded SQL simple query, because in this type of Embedded SQL a **SELECT** always returns only one row of data. However, an Embedded SQL cursor–based query can return multiple rows of data; in a cursor-based query, a DISTINCT clause returns only unique value rows.

DISTINCT and ORDER BY

The DISTINCT clause is applied before the ORDER BY clause. Therefore, the combination of DISTINCT and ORDER BY will first select an arbitrary row that satisfies the DISTINCT clause, then order those rows based on the ORDER BY clause.

DISTINCT and GROUP BY

DISTINCT and GROUP BY both group records by a specified field (or fields) and return one record for each unique value of that field. One significant difference between them is that DISTINCT calculates aggregate functions before grouping. GROUP BY calculates aggregate functions after grouping. This difference is shown in the following examples:

SQL

```
SELECT DISTINCT BY (ROUND(Age,-1)) Age,AVG(Age) AS AvgAge FROM Sample.Person
 /* AVG(Age) returns average of all ages in table */
```

SQL

```
SELECT Age,AVG(Age) AS AvgAge FROM Sample.Person GROUP BY ROUND(Age,-1)
 /* AVG(Age) returns an average age for each age group */
```

A DISTINCT clause can be specified with one or more aggregate function fields, though this is rarely meaningful because an aggregate function returns a single value. Thus the following example returns a single row:

SQL

```
SELECT DISTINCT BY (AVG(Age)) Name,Age,AVG(Age) FROM Sample.Person
```

CAUTION: If a DISTINCT clause contains aggregate functions as the only *item* or *select-item* is used with a GROUP BY clause, the DISTINCT clause is ignored. The intended combination of DISTINCT, aggregate function, and GROUP BY can be achieved using a subquery. For further details and program examples, refer to the GROUP BY clause reference page.

Letter Case and DISTINCT Optimization

DISTINCT groups together string values based on the collation type defined for the field. By default, string data type fields are defined with SQLUPPER collation, which is not case-sensitive. You can define the string collation default for the current namespace and specify a non-default field collation type when defining a field/property.

If the field/property collation type is SQLUPPER, grouped field values are returned in all uppercase letters. To group values by original letter case, or to display the returned values for a grouped field in their original letter case, use the %EXACT collation function. This is shown in the following examples, which assume that the Home_City field is defined with collation type SQLUPPER and contains the values 'New York' and 'new york':

SQL

```
SELECT DISTINCT BY (Home_City) Name,Home_City FROM Sample.Person
/* groups together Home_City values by their uppercase letter values
   returns the name of each grouped city in uppercase letters.
   Thus, 'NEW YORK' is returned.                           */
```

SQL

```
SELECT DISTINCT BY (Home_City) Name,%EXACT(Home_City) FROM Sample.Person
/* groups together Home_City values by their uppercase letter values
   returns the name of each grouped city in original letter case.
   Thus, 'New York' or 'new york' may be returned, but not both. */
```

SQL

```
SELECT DISTINCT BY (%EXACT(Home_City)) Name,Home_City FROM Sample.Person
/* groups together Home_City values by their original letter case
   returns the name of each grouped city in original letter case.
   Thus, both 'New York' and 'new york' are returned.
   Optimization is not used.                            */
```

You can optimize query performance for queries that contain a DISTINCT clause by using the Management Portal. Select **System Administration**, **Configuration**, **SQL and Object Settings**, **SQL**. View and edit the **GROUP BY and DISTINCT queries must produce original values** option. (This optimization also works for the GROUP BY clause.) The default is "No".

This default groups alphabetic values by their uppercase letter collation. This optimization takes advantage of indexes for the selected field(s). It is therefore only meaningful if an index exists for one or more of the selected fields. It collates field values as they are stored in the index; alphabetic strings are returned in all uppercase letters. You can set this system-wide option, then override it for specific queries by using the %EXACT collation function to preserve letter case.

For further details, refer to SQL and Object Settings Pages.

You can also set this option system-wide using the **$SYSTEM.SQL.Util.SetOption()** method `FastDistinct` option. To determine the current setting, call **$SYSTEM.SQL.CurrentSettings()**, which displays the `DISTINCT optimization turned on` setting; the default is 1.

Other Uses of DISTINCT

- Stream Field: DISTINCT operates on the OID of a stream field, not its actual data. Because all stream field OIDs are unique values, DISTINCT has no effect on actual stream field duplicate data values. DISTINCT BY (*StreamField*) reduces the number records where the stream field is NULL to one NULL record. For further details, see Storing and Using Stream Data (BLOBs and CLOBs).

- Asterisk Syntax: The syntax `DISTINCT *` is legal, but not meaningful, because all rows, by definition, contain some distinct unique identifier. The syntax `DISTINCT BY (*)` is not legal.

- Subquery: The use of a DISTINCT clause in a subquery is legal, but not meaningful, because a subquery returns a single value.

- No Row Data Selected: The DISTINCT clause can be used with a **SELECT** that does not access any table data. If the **SELECT** contains a FROM clause, specifying DISTINCT results in one row contain these non-table values; if you

do not specify DISTINCT (or TOP) the **SELECT** results in as many rows with identical values as the number of rows in the FROM clause table. If the **SELECT** does not contain a FROM clause, DISTINCT is legal but not meaningful. See FROM clause for more details.

- Aggregate Function: A DISTINCT clause can be used within an aggregate function to select only distinct (unique) field values for inclusion in the aggregate. Unlike the **SELECT** DISTINCT clause, DISTINCT within an aggregate function does not include NULL as a distinct (unique) value. Note that the **MAX** and **MIN** aggregate functions parse DISTINCT clause syntax without error, but this syntax performs no operation.

DISTINCT and %ROWID

Specifying the DISTINCT keyword causes a cursor-based Embedded SQL query to not set the %ROWID variable. %ROWID is not set even when DISTINCT does not limit the rows returned. This is shown in the following example:

ObjectScript

```
SET %ROWID=999
&sql(DECLARE EmpCursor CURSOR FOR
     SELECT DISTINCT Name, Home_State
     INTO :name,:state FROM Sample.Person
     WHERE Home_State %STARTSWITH 'M')
&sql(OPEN EmpCursor)
     QUIT:(SQLCODE'=0)
FOR { &sql(FETCH EmpCursor)
     QUIT:SQLCODE
     WRITE !,"RowID: ",%ROWID," row count: ",%ROWCOUNT
     WRITE " Name=",name," State=",state
}
 &sql(CLOSE EmpCursor)
```

This change of query behavior only applies to cursor-based Embedded SQL **SELECT** queries. Dynamic SQL **SELECT** queries and non-cursor Embedded SQL **SELECT** queries never set %ROWID.

DISTINCT and Transaction Processing

Specifying the DISTINCT keyword causes a query to retrieve all current data, including data that has not yet been committed by the current transaction. The transaction's READ COMMITTED isolation mode parameter (if set) is ignored; all data is retrieved in READ UNCOMMITTED mode. For further details, refer to Transaction Processing.

Examples

The following query returns one row for each distinct Home_State value:

SQL

```
SELECT DISTINCT Home_State FROM Sample.Person
ORDER BY Home_State
```

The following query returns one row for each distinct Home_State value, but returns additional fields for that row. The row that is retrieved is not predictable:

SQL

```
SELECT DISTINCT BY (Home_State) %ID,Name,Home_State,Office_State FROM Sample.Person
ORDER BY Home_State
```

The following query returns one row for each distinct combination of Home_State and Office_State values. Depending on the data, it will either return more rows or the same number of rows as the previous example:

SQL

```
SELECT DISTINCT BY (Home_State,Office_State) %ID,Name,Home_State,Office_State FROM Sample.Person
ORDER BY Home_State,Office_State
```

The following query uses DISTINCT BY to return one row for each distinct Name length:

SQL

```
SELECT DISTINCT BY ($LENGTH(Name)) Name,$LENGTH(Name) AS lname
FROM Sample.Person
ORDER BY lname
```

The following query uses DISTINCT BY to return one row for each distinct first element of FavoriteColors %List values. It lists one distinct row with FavoriteColors NULL:

SQL

```
SELECT DISTINCT BY ($LIST(FavoriteColors,1)) Name,FavoriteColors,$LIST(FavoriteColors,1) AS FirstColor
FROM Sample.Person
```

The following query returns the first 20 distinct Home_State values retrieved from Sample.Person in ascending collation sequence order. The "top" rows reflect the ORDER BY clause sequencing of all of the rows in Sample.Person.

SQL

```
SELECT DISTINCT TOP 20 Home_State FROM Sample.Person ORDER BY Home_State
```

The following query uses DISTINCT in both the main query and in a WHERE clause subquery. It returns the first 20 distinct Home_State values in Sample.Person that are also in Sample.Employee. If the subquery DISTINCT was not provided, it would retrieve the distinct Home_State values in Sample.Person that match a random selection of Home_State values in Sample.Employee:

SQL

```
SELECT DISTINCT TOP 20 Home_State FROM Sample.Person
WHERE Home_State IN(SELECT DISTINCT TOP 20 Home_State FROM Sample.Employee)
ORDER BY Home_State
```

The following query returns the first 20 distinct FavoriteColor values. This reflects the ORDER BY clause sequencing of all of the rows in Sample.Person. The FavoriteColors field is known to have NULLs, so one distinct row with FavoriteColors NULL appears at the top of the collation sequence.

SQL

```
SELECT DISTINCT BY (FavoriteColors) TOP 20 FavoriteColors,Name FROM Sample.Person
      ORDER BY FavoriteColors
```

Also note in the preceding example that because FavoriteColors is a list field, the collation sequence includes the element length byte. Thus distinct list values beginning with a three-letter element (RED) are listed before list values beginning with a four-letter element (BLUE).

See Also

- SELECT statement

- GROUP BY clause

- ORDER BY clause

- TOP clause

- Aggregate Functions

- Querying the Database

- Collation

FROM (SQL)

A SELECT clause that specifies one or more tables to query.

Synopsis

```
SELECT ... FROM [optimize-option] table-ref
    [ [AS] t-alias ]
    [, [LATERAL] table-ref [ [AS] t-alias] ] [,...]
```

Arguments

Argument	Description
optimize-hint	*Optional* — A single keyword, or a series of keywords separated by spaces, that specify query optimization options. See Specify Optimization Hints in Queries for more information.
table-ref	One or more tables, views, table-valued functions, or subqueries from which data is being retrieved, specified as a comma-separated list or with JOIN syntax. Some restrictions apply on using views with JOIN syntax. You can specify a subquery, enclosed in parentheses.
AS *t-alias*	*Optional* — An alias for the table name. Must be a valid identifier.
LATERAL	*Optional* — Enables lateral references to earlier tables in the FROM clause. See "LATERAL Keyword" for more information.

Description

The FROM clause specifies one or more tables (or views, or subqueries) from which data is queried within a SELECT statement. If no table data is being queried, the FROM clause is optional, as described below.

Multiple tables are specified as a comma-separated list, or a list separated by other JOIN syntax. Each table name can optionally be supplied an alias.

Table name aliases are used when specifying field names for multiple tables in the **SELECT** statement. If two (or more) tables are specified in the FROM clause, you indicate which table's field you want by specifying `tablename.fieldname` for each field in the **SELECT** *select-item* clause. Because table names are often long names, a short table name alias is useful in this context (`t-alias.fieldname`).

The following example show the use of table name aliases:

SQL

```
SELECT e.Name,c.Name
FROM Sample.Company AS c,Sample.Employee AS e
```

The AS keyword can be omitted. It is provided for compatibility and clarity.

Supplying a Schema Name to a Table Reference

A *table-ref* name is either qualified (schema.tablename) or unqualified (tablename). The schema name for an unqualified table name (or view name) is supplied using a schema search path or the system-wide default schema name:

1. If a schema search path is provided, InterSystems IRIS searches the specified schemas for a matching table name.

2. If a schema search path is not provided, or the schema search path does not produce a match, InterSystems IRIS uses the system-wide default schema name.

Table Joins

When you specify multiple table names in a FROM clause, InterSystems SQL performs join operations on those tables. The type of join performed is specified by a join keyword phrase or symbol between each pair of table names. When two table names are separated by a comma, a cross join is performed. For further details on the different types of joins and their syntax, refer to JOIN.

The sequence in which joins are performed is automatically determined by the SQL query optimizer and is not based on the sequence that the tables are listed in the query. If desired, you can control the sequence in which joins are performed by specifying a query optimization option.

The first two **SELECT** statements show the row counts for two individual tables, and the third example shows the row count for a **SELECT** specifying both tables. This latter results in a much larger table, a Cartesian product, where every row in the first table is matched with every row of the second table, an operation known as a Cross Join.

SQL

```
SELECT COUNT(*)
FROM Sample.Company
```

SQL

```
SELECT COUNT(*)
FROM Sample.Vendor
```

SQL

```
SELECT COUNT(*)
FROM Sample.Company,Sample.Vendor
```

You can perform the same operation using explicit CROSS JOIN syntax:

SQL

```
SELECT COUNT(*)
FROM Sample.Company CROSS JOIN Sample.Vendor
```

In most cases, the extensive data duplication of a cross join is not desirable, and some other type of join is preferable.

If you specify a WHERE clause in the **SELECT** statement, the cross join is performed, then the WHERE clause predicate(s) determine the result set. This is equivalent to performing an INNER JOIN with an ON clause. Thus the following two examples return identical results:

SQL

```
SELECT p.Name,p.Home_State,em.Name,em.Office_State
FROM Sample.Person AS p, Sample.Employee AS em
WHERE p.Name %STARTSWITH 'E' AND em.Name %STARTSWITH 'E'
```

SQL

```
SELECT p.Name,p.Home_State,em.Name,em.Office_State
FROM Sample.Person AS p INNER JOIN Sample.Employee AS em
ON p.Name %STARTSWITH 'E' AND em.Name %STARTSWITH 'E'
```

You can specify explicit join syntax (rather than using commas) in the FROM *table-ref* list to perform other types of join operations. For further details, refer to JOIN.

Query Optimization Options

By default, the InterSystems SQL query optimizer uses sophisticated and flexible algorithms to optimize the performance of complex queries involving join operations and/or multiple indexes. In most cases, these defaults provide optimal performance. However, after consultation with the InterSystems Worldwide Resource Center (WRC), you may be instructed to give "hints" to the query optimizer, specifying one or more aspects of query optimization. The *optimize-hint* argument in the FROM is used to specify these hints. You can specify multiple optimization hints in any order, separated by blank spaces. For further details, refer to Specify Optimization Hints in Queries.

You can use *optimize-hint* FROM clause keywords in a simple **SELECT** statement, in a **CREATE VIEW** view definition **SELECT** statement, or in a subquery **SELECT** statement within the FROM clause.

Table-Valued Functions in the FROM Clause

A table-valued function is a class query that is projected as a stored procedure and returns a single result set. A table-valued function is any class query which has SqlProc TRUE. A class query used as a table-valued function must be compiled in either LOGICAL or RUNTIME mode. When used as a table-valued function and compiled in RUNTIME mode, the table-valued function query will be called in LOGICAL mode.

A table-valued function follows the same naming conventions as a stored procedure name for a class query. Parameter parentheses are mandatory; the parentheses may be empty, enclose a literal or a host variable, or a comma-separated list of literals and host variables. If you specify no parameters (empty parentheses or the null string), the table-valued function returns all data rows.

To issue a query using a table-valued function, the user must hold the EXECUTE privilege on the stored procedure that defines the table-valued function. The user must also have SELECT privileges on the tables or views accessed by the table-valued function query.

In the following example, the class query Sample.Person.ByName is projected as a stored procedure and can thus be used as a table-valued function:

SQL

```
SELECT Name,DOB FROM Sample.SP_Sample_By_Name('A')
```

The following Dynamic SQL example specifies the same table-valued function. It uses the **%Execute()** method to supply parameter values to the ? input parameter:

ObjectScript

```
SET myquery="SELECT Name,DOB FROM Sample.SP_Sample_By_Name(?)"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute("A")
DO rset.%Display()
WRITE !,"End of A data",!!
SET rset = tStatement.%Execute("B")
DO rset.%Display()
WRITE !,"End of B data"
```

A table-valued function can only be used in the FROM clause of either a **SELECT** statement or a DECLARE statement. A table-valued function name can be qualified with a schema name or unqualified (without a schema name); an unqualified name uses the default schema. In a **SELECT** statement FROM clause, a table-valued function can be used wherever a table name can be used. It can be used in a view or a subquery, and can be joined to other *table-ref* items using a comma-separated list or explicit **JOIN** syntax.

A table-valued function cannot be directly used in an **INSERT**, **UPDATE**, or **DELETE** statement. You can, however, specify a subquery for these commands that specifies a table-valued function.

InterSystems SQL does not define the EXTENTSIZE for a table-valued function, or the SELECTIVITY for table-valued function columns.

Subqueries in the FROM Clause

You can specify a subquery in the FROM clause. This is known as a streamed subquery. The subquery is treated the same as a table, including its use in JOIN syntax and the optional assignment of an alias using the AS keyword. A FROM clause can contain multiple tables, views, and subqueries in any combination, subject to the restrictions of the JOIN syntax, as described in JOIN.

A subquery is enclosed in parentheses. The following example shows a subquery in a FROM clause:

SQL

```
SELECT name,region
FROM (SELECT t1.name,t1.state,t2.region
      FROM Employees AS t1 LEFT OUTER JOIN Regions AS t2
      ON t1.state=t2.state)
GROUP BY region
```

A subquery can specify a TOP clause. A subquery can contain an ORDER BY clause when paired with a TOP clause.

A subquery can use SELECT * syntax, subject to the following restriction: because a FROM clause results in a value expression, a subquery containing SELECT * must yield only one column.

A join within a subquery cannot be a NATURAL join or take a USING clause.

FROM Subqueries and %VID

When a FROM subquery is invoked, it returns a %VID for each subquery row returned. A %VID is an integer counter field; its values are system-assigned, unique, non-null, non-zero, and non-modifiable. The %VID is only returned when explicitly specified. It is returned as data type INTEGER. Because %VID values are sequential integers, they are far more meaningful if the subquery returns ordered data; a subquery can only use an ORDER BY clause when it is paired with a TOP clause.

Because the %VID is a sequential integer, it can be used to determine the ranking of items in a subquery with an ORDER BY clause. In the following example, the 10 newest records are listed in Name order, but their timestamp ranking is easily seen using the %VID values:

SQL

```
SELECT Name,%VID,TimeStamp FROM
   (SELECT TOP 10 * FROM MyTable ORDER BY TimeStamp DESC)
ORDER BY Name
```

One common use of the %VID is to "window" the result set, dividing execution into sequential subsets that fit the number of lines available in a display window. For example, display 20 records, then wait for the user to press Enter, then display the next 20 records.

The following example uses %VID to "window" the results into subsets of 10 records:

SQL

```
SELECT %VID,* FROM
   (SELECT TOP 60 Name, Age FROM Sample.Person WHERE Age > 55 ORDER BY Name)
WHERE %VID BETWEEN ? AND ?
```

For details on using %VID, refer to Defining and Using Views.

LATERAL Keyword

The LATERAL keyword can be used to have more explicit control over the FROM processing order by allowing views and table-valued functions to reference field values from tables listed earlier in the FROM clause. These lateral references affect the rows that the subquery or table valued-function generate.

Within the subquery or the table-valued function that the LATERAL keyword is specified on, the laterally referenced fields are treated as given values. Laterally referenced fields always come from earlier FROM items in the same FROM clause.

When used to precede a FROM subquery, the LATERAL keyword indicates that the subquery may reference fields in FROM items that semantically precede it in the query. These laterally referenced fields will be processed before the FROM subquery that LATERAL was specified on.

When used to precede a table-valued function, the LATERAL keyword indicates that fields from the previous FROM items can be used within the table-valued function. In this context, the keyword is optional, and the lateral join will be applied implicitly if such references are used within the table-valued function.

Optional FROM Clause

If no table data is referenced (directly or indirectly) by the **SELECT** item list, the FROM clause is optional. This kind of **SELECT** may be used to return data from functions, operator expressions, constants, or host variables. For a query that references no table data:

- If the FROM clause is omitted, a maximum of one row of data is returned, regardless of the TOP keyword value; TOP 0 returns no data. The DISTINCT clause is ignored. No privileges are required.

- If the FROM clause is specified, it must specify an existing table in the current namespace. You must have SELECT privilege for that table, even though the table is not referenced. The number of identical rows of data returned is equal to the number of rows in the specified table, unless you specify a TOP or DISTINCT clause, or limit it with a WHERE or HAVING clause. Specifying a DISTINCT clause limits the output to a single row of data. The TOP keyword limits the output to the number of rows specified by the TOP value; TOP 0 returns no data.

With or without a FROM clause, subsequent clauses (WHERE, GROUP BY, HAVING or ORDER BY) may be specified. A WHERE or HAVING clause may be used to determine whether or not to return results, or how many identical rows of results to return. These clauses may reference a table, even if no FROM clause is specified. A GROUP BY or ORDER BY clause may be specified, but these clauses are not meaningful.

The following are examples of **SELECT** statements that reference no table data. Both examples return one row of information.

The following example omits the FROM clause. The DISTINCT keyword is not needed, but may be specified. No SELECT clauses are permitted.

SQL

```
SELECT 3+4 AS Arith,
       {fn NOW} AS NowDateTime,
       {fn DAYNAME({fn NOW})} AS NowDayName,
       UPPER('MixEd cASe EXPreSSioN') AS UpCase,
       {fn PI} AS PiConstant
```

The following example includes a FROM clause. The DISTINCT keyword is used to return a single row of data. The FROM clause table reference must be a valid table. The ORDER BY clause is permitted here, but is meaningless. Note that the ORDER BY clause must specify a valid select item alias:

SQL

```
SELECT DISTINCT 3+4 AS Arith,
    {fn NOW} AS NowDateTime,
    {fn DAYNAME({fn NOW})} AS NowDayName,
    UPPER('MixEd cASe EXPreSSioN')  AS UpCase,
    {fn PI} AS PiConstant
FROM Sample.Person
ORDER BY NowDateTime
```

The following examples both use a WHERE clause to determine whether or not to return results. The first includes a FROM clause and uses the DISTINCT keyword is to return a single row of data. The second omits the FROM clause, and therefore returns at most a single row of data. In both cases, the WHERE clause table reference must be a valid table for which you have SELECT privilege:

SQL

```
SELECT DISTINCT
    {fn NOW} AS DataOKDate
FROM Sample.Person
WHERE FOR SOME (Sample.Person)(Name %STARTSWITH 'A')
```

SQL

```
SELECT {fn NOW} AS DataOKDate
WHERE FOR SOME (Sample.Person)(Name %STARTSWITH 'A')
```

See Also

- SELECT

- JOIN

- Querying the Database

- Defining Tables

- Analyze SQL Statements and Statistics

- SQLCODE error messages

GROUP BY (SQL)

A SELECT clause that groups the resulting rows of a query according to one or more columns.

Synopsis

```
SELECT ...
GROUP BY field {,field2}
```

Description

GROUP BY is a clause of the SELECT command. The optional **GROUP BY** clause appears after the **FROM** clause and the optional **WHERE** clause, and before the optional **HAVING** and **ORDER BY** clauses.

The **GROUP BY** clause takes the resulting rows of a query and breaks them up into individual groups according to one or more database columns. When you use **SELECT** in conjunction with **GROUP BY**, one row is retrieved for each distinct value of the **GROUP BY** fields. **GROUP BY** treats fields with NULL (no specified value) as a separate distinct value group.

The **GROUP BY** clause is conceptually similar to the InterSystems IRIS aggregate function extension keyword %FOREACH, but **GROUP BY** operates on an entire query, while %FOREACH allows selection of aggregates on sub-populations without restricting the entire query population.

GROUP BY can be used in the SELECT clause of an INSERT command. **GROUP BY** cannot be used in an **UPDATE** or **DELETE** command.

Specifying a Field

The simplest form of a **GROUP BY** clause specifies a single field, such as GROUP BY City. This selects one arbitrary row for each unique City value. You can also specify a comma-separated list of fields, the combined value of which is treated as a single grouping term. It selects one arbitrary row for each unique combination of City and Age values. Therefore, GROUP BY City,Age returns the same results as GROUP BY Age,City.

The field(s) must be specified by column name. Valid *field* values include the following: a column name (GROUP BY City); an %ID (which returns all rows); a scalar function specifying a column name (GROUP BY ROUND(Age,-1)); a collation function specifying a column name (GROUP BY %EXACT(City)).

You cannot specify a field by column alias; attempting to do so generates an SQLCODE -29 error. You cannot specify a field by column number; this is interpreted as a literal and returns one row. You cannot specify an aggregate field; attempting to do so generates an SQLCODE -19 error. You cannot specify a subquery; this is interpreted as a literal and returns one row.

GROUP BY *StreamField* operates on the OID of a stream field, not its actual data. Because all stream field OIDs are unique values, **GROUP BY** has no effect on actual stream field duplicate data values. **GROUP BY** *StreamField* reduces the number records where the stream field is NULL to one record. For further details, see Storing and Using Stream Data (BLOBs and CLOBs).

A **GROUP BY** clause can use the arrow syntax (–>) operator to specify a field in a table that is not the base table. For example: GROUP BY Company->Name. For further details, refer to Implicit Joins (Arrow Syntax).

Specifying a literal as the *field* value in a **GROUP BY** clause returns 1 row; which row is returned is indeterminate. Thus, specifying 7, 'Chicago', '', 0, or NULL all return 1 row. However, if you specify a literal as a *field* value in a comma-separated list, the literal is ignored and **GROUP BY** selects one arbitrary row for each unique combination of the specified field names.

Aggregate Functions with GROUP BY and DISTINCT BY

The **GROUP BY** clause is applied before aggregate functions are calculated. In the following example, the **COUNT** aggregate function counts the number of rows in each **GROUP BY** group:

SQL

```
SELECT Home_State,COUNT(Home_State)
FROM Sample.Person
GROUP BY Home_State
```

The DISTINCT BY clause is applied after aggregate functions are calculated. In the following example, the **COUNT** aggregate function counts the number of rows in the entire table:

SQL

```
SELECT DISTINCT BY(Home_State) Home_State,COUNT(Home_State)
FROM Sample.Person
```

In order to calculate an aggregate function for the entire table, rather than a **GROUP BY** group, you can specify a *select-item* subquery:

SQL

```
SELECT Home_State,(SELECT COUNT(Home_State) FROM Sample.Person)
FROM Sample.Person
GROUP BY Home_State
```

A **GROUP BY** clause should not be used with a DISTINCT clause when the select list consists of an aggregate field. For example, the following query is *intended* to return the distinct numbers of people who share the same Home_State:

SQL

```
/* This query DOES NOT apply the DISTINCT keyword */
/* It is provided as a cautionary example  */
SELECT DISTINCT COUNT(*) AS mynum
FROM Sample.Person
GROUP BY Home_State
ORDER BY mynum
```

This query did not return the expected results because it did not apply the DISTINCT keyword. To apply both a **DISTINCT** aggregate and a **GROUP BY** clause, use a subquery as shown in the following example:

SQL

```
SELECT DISTINCT *
FROM (SELECT COUNT(*) AS mynum
      FROM Sample.Person
      GROUP BY Home_State) AS Sub
ORDER BY Sub.mynum
```

This example successfully returns the distinct numbers of people who share the same Home_State. For instance, if any Home_State is shared by 8 people, the query returns an 8.

If a query consist only of aggregate functions and does not return any data from the table, it returns %ROWCOUNT=1 with an empty string (or 0) value for the aggregate functions. For example:

SQL

```
SELECT AVG(Age) FROM Sample.Person WHERE Name %STARTSWITH 'ZZZZ'
```

However, if this type of query contains a **GROUP BY** clause, it returns %ROWCOUNT=0 and the aggregate function values remains undefined.

Collation, Letter Case, and Optimization

This section describes how **GROUP BY** handles data values that differ only in letter case.

- Group Lettercase Variants Together (return uppercase):

  By default, **GROUP BY** groups together string values based on the collation specified for the *field* when it was created. InterSystems IRIS has a default string collation, which can be set for each namespace; the initial string collation default for all namespaces is SQLUPPER. Therefore, commonly, **GROUP BY** collation is not case-sensitive unless otherwise specified.

  GROUP BY groups together the values of a field with SQLUPPER collation based on their uppercase letter collation. Field values that differ only in letter case are grouped together. Grouped field values are returned in all uppercase letters. This has the performance advantage of allowing **GROUP BY** to use the index for the field, rather than accessing the actual field values. It is therefore only meaningful if an index exists for one or more of the selected fields. It has the consequence that the **GROUP BY** field value is returned in all uppercase letters, even if none of the actual data values are in all uppercase letters.

- Group Lettercase Variants Together (return actual lettercase):

  GROUP BY can group together values that differ in lettercase and return grouped field values with an actual field lettercase value (randomly selecting). This has the advantage that the returned value is an actual value, showing the lettercase of at least one value in the data. It has the performance disadvantage of not being able to use the field's index. You can specify this for an individual query by applying the %EXACT collation function to the select-item field.

- Do Not Group Lettercase Variants Together (return actual lettercase):

  GROUP BY can perform case-sensitive grouping of values by applying the %EXACT collation function to the **GROUP BY** field. This has the advantage of returning every lettercase variant as a separate group. It has the performance disadvantage of not being able to use the field's index.

You can configure this behavior system-wide for all queries that contain a **GROUP BY** clause by using the Management Portal. Select **System Administration**, **Configuration**, **SQL and Object Settings**, **SQL**. View and edit the **GROUP BY and DISTINCT queries must produce original values** check box. By default, this check box is not selected. This default groups alphabetic values by their uppercase letter collation. (This optimization also works for the DISTINCT clause.) For further details, refer to SQL and Object Settings Pages.

You can also set this option system-wide using the **$SYSTEM.SQL.Util.SetOption()** method `FastDistinct` option. To determine the current setting, call **$SYSTEM.SQL.CurrentSettings()**, which displays the `DISTINCT optimization turned on` setting; the default is 1.

This optimization takes advantage of indexes for the selected field(s). It is therefore only meaningful if an index exists for one or more of the selected fields. It collates field values as they are stored in the index; alphabetic strings are returned in all uppercase letters. You can set this system-wide option, then override it for specific queries by using the %EXACT collation function to preserve letter case.

The following examples show these behaviors. These examples assume that Sample.Person contains records with a Home_City field with SQLUPPER collation and values of 'New York' and 'new york':

SQL

```
SELECT Home_City FROM Sample.Person GROUP BY Home_City
/* groups together Home_City values by their uppercase letter values
   returns the name of each grouped city in uppercase letters.
   Thus, 'NEW YORK' is returned.                              */
```

SQL

```
SELECT %EXACT(Home_City) FROM Sample.Person GROUP BY Home_City
/* groups together Home_City values by their uppercase letter values
   returns the name of a grouped city in original letter case.
   Thus, 'New York' or 'new york' may be returned, but not both. */
```

SQL

```
SELECT Home_City FROM Sample.Person GROUP BY %EXACT(Home_City)
/* groups together Home_City values by their original letter case
   returns the name of each grouped city in original letter case.
   Thus, both 'New York' and 'new york' are returned as separate groups. */
```

%ROWID

Specifying a GROUP BY clause causes a cursor-based Embedded SQL query to not set the %ROWID variable. %ROWID is not set even when GROUP BY does not limit the rows returned. This is shown in the following example:

ObjectScript

```
SET %ROWID=999
&sql(DECLARE EmpCursor CURSOR FOR
     SELECT Name, Home_State
     INTO :name,:state FROM Sample.Person
     WHERE Home_State %STARTSWITH 'M'
     GROUP BY Home_State)
&sql(OPEN EmpCursor)
     QUIT:(SQLCODE'=0)
FOR { &sql(FETCH EmpCursor)
     QUIT:SQLCODE
     WRITE !,"RowID: ",%ROWID," row count: ",%ROWCOUNT
     WRITE " Name=",name," State=",state
}
&sql(CLOSE EmpCursor)
```

This change of query behavior only applies to cursor-based Embedded SQL **SELECT** queries. Dynamic SQL **SELECT** queries and non-cursor Embedded SQL **SELECT** queries never set %ROWID.

Transaction Committed Changes

A query containing a **GROUP BY** clause does not support READ COMMITTED isolation level. In a transaction defined as READ COMMITTED, a **SELECT** statement without a **GROUP BY** clause returns only data modifications that have been committed; in other words, it returns the state of the data before the current transaction. A **SELECT** statement with a **GROUP BY** clause returns all data modifications made, whether or not they have been committed.

Arguments

field

One or more fields from which data is being retrieved. Either a single field name or a comma-separated list of field names.

Example

The following example groups names by their initial letter. It returns the initial letter, the count of names sharing that initial letter, and an example of a one of the name values. Names are grouped using their SQLUPPER collation, regardless of the letter case of the actual values. Note that the Name select-item contains the uppercase initial letter; %EXACT collation is used to display an actual name value:

SQL

```
SELECT Name AS Initial,COUNT(Name) AS SameInitial,%EXACT(Name) AS Example
FROM Sample.Person GROUP BY %SQLUPPER(Name,2)
```

See Also

- SELECT

- DISTINCT clause

- JOIN

HAVING (SQL)

A SELECT clause that specifies one or more restrictive conditions on a group of data values.

Synopsis

```
SELECT field
    FROM table
    GROUP BY field
    HAVING condition-expression

SELECT aggregatefunc(field %AFTERHAVING)
FROM table
[GROUP BY field]
HAVING condition-expression
```

Description

The optional **HAVING** clause appears after the **FROM** clause and the optional **WHERE** and **GROUP BY** clauses, and before the optional **ORDER BY** clause.

The **HAVING** clause of a SELECT statement qualifies or disqualifies specific rows from the query selection. The rows that qualify are those for which the *condition-expression* is true. The *condition-expression* is a series of logical tests (predicates) which can be linked by the AND and OR logical operators. For further details, see the WHERE clause.

The **HAVING** clause is like a **WHERE** clause that can operate on groups, rather than on the full data set. Thus, in most cases, the **HAVING** clause is used either with an aggregate function using the %AFTERHAVING keyword, or in combination with a GROUP BY clause, or both.

A **HAVING** clause *condition-expression* can also specify an aggregate function. A **WHERE** clause *condition-expression* cannot specify an aggregate function. This is shown in the following example:

SQL

```
SELECT Name,Age,AVG(Age) AS AvgAge
FROM Sample.Person
HAVING Age > AVG(Age)
ORDER BY Age
```

A **HAVING** clause often serves to compare aggregates of sub-populations against aggregates for an entire population.

Specifying a Field

A field specified in a **HAVING** clause *condition-expression* or an %AFTERHAVING keyword expression must be specified as a field name or an aggregate function. You cannot specify a field or aggregate function by column number. You cannot specify a field or aggregate function by column alias; attempting to do so generates an SQLCODE -29 error. However, you can use a subquery to define a column alias, then use this alias in the **HAVING** clause. For example:

SQL

```
SELECT Y AS TeenYear,AVG(Y %AFTERHAVING) AS AvgTeenAge FROM
    (SELECT Age AS Y FROM Sample.Person WHERE Age<20)
HAVING Y > 12 ORDER BY Y
```

Aggregate Functions in the select-item List

The **HAVING** clause selects which rows to return. By default, this row selection does not determine the value of aggregate functions in the *select-item* list because the **HAVING** clause is parsed after aggregate functions in the *select-item* list.

In the following example, only those rows with Age > 65 are returned. But the AVG(Age) is calculated based on all rows, not just those selected by the **HAVING** clause:

SQL

```
SELECT Name,Age,AVG(Age) AS AvgAge FROM Sample.Person
HAVING Age > 65
 ORDER BY Age
```

Compare this to a **WHERE** clause, which selects both which rows to return and which row values to include in aggregate functions in the *select-item* list:

SQL

```
SELECT Name,Age,AVG(Age) AS AvgAge FROM Sample.Person
WHERE Age > 65
ORDER BY Age
```

A **HAVING** clause can be used in a query that *only* returns aggregate values:

- Aggregate Threshold: The **HAVING** clause uses an aggregate threshold to determine whether to return 1 row (containing the query aggregate values) or 0 rows. Thus you can use a **HAVING** clause to only return an aggregate calculation when an aggregate threshold is achieved. The following example only returns an average of the *Age* values for all rows in the table when there are at least 100 rows in the table. If there are less than 100 rows, the average of the *Age* values for all rows might not be deemed meaningful, and therefore should not be returned:

  SQL

  ```
  SELECT AVG(Age) FROM Sample.Person HAVING COUNT(*)>99
  ```

- Multiple Rows: A **HAVING** clause with an aggregate function and no **GROUP BY** clause returns the number of rows that fulfill the **HAVING** clause condition. The aggregate function value is calculated based on all of the rows in the table:

  SQL

  ```
  SELECT AVG(Age) FROM Sample.Person HAVING %ID<10
  ```

  This is in contrast to a **WHERE** clause with an aggregate function, which returns one row. The aggregate function value is calculated based on rows that fulfill the **WHERE** clause condition:

  SQL

  ```
  SELECT AVG(Age) FROM Sample.Person WHERE %ID<10
  ```

%AFTERHAVING

The %AFTERHAVING keyword can be used with an aggregate function in the *select-item* list to specify that the aggregate operation is to be performed after the **HAVING** clause condition is applied.

SQL

```
SELECT Name,Age,AVG(Age) AS AvgAge,
 AVG(Age %AFTERHAVING) AS AvgMiddleAge
 FROM Sample.Person
 HAVING Age > 40 AND Age < 65
 ORDER BY Age
```

The %AFTERHAVING keyword only gives meaningful results if both of the following considerations are met:

- The *select-item* list must contain at least one item that is a non-aggregate field reference. This field reference may be to any field in any table specified in the **FROM** clause, a field referenced using an implicit join (arrow syntax), the %ID alias, or an asterisk (*).

- The **HAVING** clause condition must apply at least one non-aggregate condition. Therefore, `HAVING Age>50`, `HAVING Age>AVG(Age)`, or `HAVING Age>50 AND MAX(Age)>75` are valid conditions, but `HAVING Age>50 OR MAX(Age)>75` is not a valid condition.

The following example uses a **HAVING** clause with a **GROUP BY** clause to return the state average age, and the state average age for people that are older than the average age for all rows in the table. It also uses a subquery to return the average age for all rows in the table:

SQL

```
SELECT Home_State,(SELECT AVG(Age) FROM Sample.Person) AS AvgAgeAllRecs,
       AVG(Age) AS AvgAgeByState,AVG(Age %AFTERHAVING) AS AvgOlderByState
FROM Sample.Person
GROUP BY Home_State
HAVING Age > AVG(Age)
ORDER BY Home_State
```

Arguments

condition-expression

An expression consisting of one or more boolean predicates governing which data values are to be retrieved.

Logical Predicates

The SQL predicates fall into the following categories:

- Equality Comparison Predicates

- BETWEEN Predicate

- IN and %INLIST Predicates

- %STARTSWITH Predicate

- Contains Operator ([)

- FOR SOME Predicate

- NULL Predicate

- EXISTS Predicate

- LIKE, %MATCHES, and %PATTERN Predicates

- %INSET and %FIND Predicates

Note: You cannot use the **FOR SOME %ELEMENT** collection predicate in a **HAVING** clause. This predicate can only be used in a **WHERE** clause.

Predicate Case-Sensitivity

A predicate uses the collation type defined for the field. By default, string data type fields are defined with SQLUPPER collation, which is not case-sensitive. You can define the string collation default for the current namespace and specify a non-default field collation type when defining a field/property.

The **%INLIST**, Contains operator ([), **%MATCHES**, and **%PATTERN** predicates do not use the field's default collation. They always uses EXACT collation, which is case-sensitive.

A predicate comparison of two literal strings is always case-sensitive.

Predicate Conditions and %NOINDEX

You can preface a predicate condition with the %NOINDEX keyword to prevent the query optimizer using an index on that condition. This is most useful when specifying a range condition that is satisfied by the vast majority of the rows. For example, `HAVING %NOINDEX Age >= 1`. For further details, refer to Index Optimization Options.

Equality Comparison Predicates

The following are the available comparison predicates:

Table C–1: SQL Equality Comparison Predicates

Predicate	Operation
=	Equals
<>	Does not equal
!=	Does not equal
>	Is greater than
<	Is less than
>=	Is greater than or equal to
<=	Is less than or equal to

The following example uses a comparison predicate. It returns one record for each Age less than 21:

SQL

```
SELECT Name, Age FROM Sample.Person
GROUP BY Age
HAVING Age < 21
ORDER BY Age
```

Note that SQL defines comparison operations in terms of collation: the order in which values are sorted. Two values are equal if they collate in exactly the same way. A value is greater than another value if it collates after the second value. String data type field collation is based on the field's default collation. By default, it is not case-sensitive. Thus, a comparison of two string field values or a comparison of a string field value with a string literal is (by default) not case-sensitive. For example, if Home_State field values are uppercase two-letter strings:

Expression	Value
'MA' = Home_State	TRUE for values MA.
'ma' = Home_State	TRUE for values MA.
'VA' < Home_State	TRUE for values VT, WA, WI, WV, WY.
'ar' >= Home_State	TRUE for values AK, AL, AR.

Note, however, that a comparison of two literal strings *is* case-sensitive: `WHERE  'ma'='MA'` is always FALSE.

BETWEEN Predicate

The following example uses a BETWEEN predicate, which is equivalent to a paired greater than or equal to and less than or equal to. It returns one record for each Age between 18 and 35, inclusive of 18 and 35:

SQL

```
SELECT Name, Age FROM Sample.Person
GROUP BY Age
HAVING Age BETWEEN 18 AND 35
ORDER BY Age
```

For further details, refer to BETWEEN.

IN and %INLIST Predicates

The IN predicate is used for matching a value to an unstructured series of items.

The %INLIST predicate is an InterSystems IRIS extension for matching a value to the elements of a list structure.

With either predicate you can perform equality comparisons and subquery comparisons.

IN has two formats. The first serves as shorthand for the use of multiple equality comparisons linked together with the OR operator. For instance:

SQL

```
SELECT Name, Home_State FROM Sample.Person
GROUP BY Home_State
HAVING Home_State IN ('ME','NH','VT','MA','RI','CT')
```

evaluates true if Home_State equals any of the values inside the parenthetical list. The list elements can be constants or expressions. Collation applies to the IN comparison as it applies to an equality test. By default, IN comparisons use the collation type of the field definition; by default string fields are defined as SQLUPPER, which is not case-sensitive.

When dates or times are used for IN predicate equality comparisons, the appropriate data type conversions are automatically performed. If the **HAVING** clause field is type TimeStamp, values of type Date or Time are converted to Timestamp. If the **HAVING** clause field is type Date, values of type TimeStamp or String are converted to Date. If the **HAVING** clause field is type Time, values of type TimeStamp or String are converted to Time.

The following examples both perform the same equality comparisons and return the same data. The **GROUP BY** field specifies to return only one record for each successful equality comparison. The DOB field is of data type Date:

SQL

```
SELECT Name,DOB FROM Sample.Person
GROUP BY DOB
HAVING DOB IN ({d '1951-02-02'},{d '1987-02-28'})
```

SQL

```
SELECT Name,DOB FROM Sample.Person
GROUP BY DOB
HAVING DOB IN ({ts '1951-02-02 02:37:00'},{ts '1987-02-28 16:58:10'})
```

For further details refer to Date and Time Constructs.

The **%INLIST** predicate can be used to perform an equality comparison on the elements of a list structure. **%INLIST** uses EXACT collation. Therefore, by default, **%INLIST** string comparisons are case-sensitive. For further details on list structures, see the SQL $LIST function.

The following example uses %INLIST to match a string value to the elements of the FavoriteColors list field:

SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
HAVING 'Red' %INLIST FavoriteColors
```

It returns all records where FavoriteColors includes the element "Red".

The following example matches Home_State column values to the elements of the *northne* (northern New England states) list:

SQL

```
SELECT Name,Home_State
FROM Sample.Person
HAVING Home_State %INLIST $LISTBUILD("VT","NH","ME")
```

You can also use IN or %INLIST with a subquery to test whether a column value (or any other expression) equals any of the subquery row values. For example:

SQL

```
SELECT Name,Home_State FROM Sample.Person
HAVING Name IN
 (SELECT Name FROM Sample.Employee
 HAVING Salary < 50000)
```

Note that the subquery must have exactly one item in the SELECT list.

For further details, refer to IN and %INLIST.

%STARTSWITH Predicate

The InterSystems IRIS **%STARTSWITH** comparison operator permits you to perform partial matching on the initial characters of a string or numeric. The following example uses **%STARTSWITH**. It selects by age, then returns a record for each Name that begins with "S":

SQL

```
SELECT Name,Age FROM Sample.Person
WHERE Age > 30
HAVING Name %STARTSWITH 'S'
ORDER BY Name
```

Like other string field comparisons, **%STARTSWITH** comparisons are not case-sensitive. For further details, refer to %STARTSWITH.

Contains Operator ([)

The Contains operator is the open bracket symbol: [. It permits you to match a substring (string or numeric) to any part of a field value. The comparison is always case-sensitive. The following example uses the Contains operator in a HAVING clause to select those records in which the Home_State value contains a "K", and then do an %AFTERHAVING count on those states:

SQL

```
SELECT Home_State,COUNT(Home_State) AS States,
   COUNT(Home_State %AFTERHAVING) AS KStates
 FROM Sample.Person
 HAVING Home_State [ 'K'
```

FOR SOME Predicate

The FOR SOME predicate of the HAVING clause determines whether or not to return a result set based on a condition test of one or more field values. This predicate has the following syntax:

```
FOR SOME (table[AS t-alias]) (fieldcondition)
```

FOR SOME specifies that *fieldcondition* must evaluate to true; at least one of the field values must match the specified condition. *table* can be a single table or a comma-separated list of tables, and can optionally take a table alias. *fieldcondition*

specifies one or more conditions for one or more fields within the specified *table*. Both the *table* argument and the *fieldcondition* argument must be delimited by parentheses.

The following example shows the use of the FOR SOME predicate:

SQL

```
SELECT Name,Age
FROM Sample.Person
HAVING FOR SOME (Sample.Person)(Age>20)
ORDER BY Age
```

In the above example, if at least one field contains an Age value greater than 20, all of the records are returned. Otherwise, no records are returned.

For further details, refer to FOR SOME.

NULL Predicate

This detects undefined values. You can detect all null values, or all non-null values:

SQL

```
SELECT Name, FavoriteColors FROM Sample.Person
HAVING FavoriteColors IS NULL
```

SQL

```
SELECT Name, FavoriteColors FROM Sample.Person
HAVING FavoriteColors IS NOT NULL
ORDER BY FavoriteColors
```

Using the **GROUP BY** clause, you can return one record for each non-null value for a specified field:

SQL

```
SELECT Name, FavoriteColors FROM Sample.Person
GROUP BY FavoriteColors
HAVING FavoriteColors IS NOT NULL
ORDER BY FavoriteColors
```

For further details, refer to NULL.

EXISTS Predicate

This operates with subqueries to test whether a subquery evaluates to the empty set.

SQL

```
SELECT t1.disease FROM illness_tab t1 WHERE EXISTS
 (SELECT t2.disease FROM disease_registry t2
 WHERE t1.disease = t2.disease
 HAVING COUNT(t2.disease) > 100)
```

For further details, refer to EXISTS.

LIKE, %MATCHES, and %PATTERN Predicates

These three predicates allow you to perform pattern matching.

- LIKE allows you to pattern match using literals and wildcards. Use LIKE when you wish to return data values that contain a known substring of literal characters, or contain several known substrings in a known sequence. LIKE uses the collation of its target for letter case comparisons.

- **%MATCHES** allows you to pattern match using literals, wildcards, and lists and ranges. Use %MATCHES when you wish to return data values that contain a known substring of literal characters, or contain one or more literal characters that fall within a list or range of possible characters, or contain several such substrings in a known sequence. %MATCHES uses EXACT collation for letter case comparisons.

- **%PATTERN** allows you to specify a pattern of character types. For example, `'1U4L1",".A'` (1 uppercase letter, 4 lowercase letters, one literal comma, followed by any number of letter characters of either case). Use %PATTERN when you wish to return data values that contain a known sequence of character types. %PATTERN is especially useful when the data value is unimportant, but the character type format of those values is significant. %PATTERN can also specify known literal characters. It uses EXACT collation for literal comparisons, which are always case-sensitive.

To perform a comparison with the first characters of a string, use the %STARTSWITH predicate.

Examples

The following example returns a row for each state that has at least one person under the age of 21. For each row it returns the average, minimum, and maximum ages of all people in the state.

SQL

```
SELECT Home_State, MIN(Age) AS Youngest,
  AVG(Age) AS AvgAge, MAX(Age) AS Oldest
 FROM Sample.Person
 GROUP BY Home_State
 HAVING Age < 21
 ORDER BY Youngest
```

The following example returns a row for each state that has at least one person under the age of 21. For each row it returns the average, minimum, and maximum ages of all people in the state. Using the %AFTERHAVING keyword, it also returns the average age of those people in the state under the age of 21 (AvgYouth), and the age of the oldest person in the state under the age of 21 (OldestYouth).

SQL

```
SELECT Home_State,AVG(Age) AS AvgAge,
   AVG(Age %AFTERHAVING) AS AvgYouth,
   MIN(Age) AS Youngest, MAX(Age) AS Oldest,
   MAX(Age %AFTERHAVING) AS OldestYouth
 FROM Sample.Person
 GROUP BY Home_State
 HAVING Age < 21
 ORDER BY AvgAge
```

For further examples of %AFTERHAVING, refer to the individual aggregate functions.

See Also

- SELECT statement

- WHERE clause

- GROUP BY clause

- Overview of Predicates

- Querying the Database

INTO (SQL)

A SELECT clause that specifies the storing of selected values in host variables.

Synopsis

```
INTO :hostvar1 [,:hostvar2]...
```

Description

The **INTO** clause and host variables are only used in Embedded SQL. They are not used in Dynamic SQL. In Dynamic SQL, similar functionality for output variables is provided by the %SQL.Statement class. Specifying an **INTO** clause in a SELECT query processed via ODBC, JDBC, or Dynamic SQL results in an SQLCODE -422 error.

An **INTO** clause can be used in a SELECT, DECLARE, or FETCH statement. The **INTO** clause is identical for all three statements; examples on this page all refer to the **SELECT** statement. For usage with **DECLARE** and **FETCH**, refer to SQL Cursors.

The **INTO** clause uses the values retrieved (or calculated) in the **SELECT** *select-item* list to set corresponding output host variables, making these returned data values available to ObjectScript. In a **SELECT** the optional **INTO** clause appears after the *select-item* list and before the **FROM** clause.

CAUTION:	Output host variables are initialized to the empty string when Embedded SQL is compiled. This prevents <UNDEFINED> errors at execution time. Therefore, host variables only contain meaningful values when SQLCODE=0. Always check SQLCODE before using an output host variable value. Do not use these variable values when SQLCODE=100 or when SQLCODE is a negative number.

Host Variables

A host variable can contain only a single value. Therefore, a **SELECT** in embedded SQL only retrieves one row of data. This defaults to the first row of the table. You can, of course, retrieve data from some other row of the table by limiting the eligible rows using a **WHERE** condition.

In embedded SQL you can return data from multiple rows by declaring a cursor and then issuing a **FETCH** for each successive row. The **INTO** clause host variables can be specified in the **DECLARE** query or specified in the **FETCH**.

INTO clause host variables can be specified in either of two ways (or a combination of both):

- A host variable list, consisting of a comma-separated list of host variables, one for each *select-item*.
- A host variable array, consisting of a single subscripted host variable.

For important restrictions on the use of host variable values in the containing program, refer to Host Variables.

Note:	If the host language declares data types for variables, all host variables must be declared in the host language before invoking the **SELECT** statement. The data types of the retrieved field values must match the host variable declarations. (ObjectScript does not declare data types for variables.)

Using a Host Variable List

The following rules apply when you specify a host variable list in the **INTO** clause:

- The number of host variables in the **INTO** clause must match the number of fields specified in the *select-item* list. If the number of selected fields and host variables differs, SQL returns a "cardinality mismatch" error.
- Selected fields and host variables are matched by relative position. Therefore, the corresponding items in these two lists must appear in the same sequence.

- The listed host variables may be any combination of unsubscripted or subscripted variables.

- A listed host variable can return an aggregate value (such as a count, sum, or average) or a function value.

- A listed host variable can return %CLASSNAME and %TABLENAME values.

- Listed host variables can return field values from a SELECT involving multiple tables, or return values from a SELECT with no FROM clause.

The following example selects four fields into a list of four host variables. The host variables in this example are subscripted:

ObjectScript

```
&sql(SELECT %ID,Home_City,Name,SSN
     INTO :mydata(1),:mydata(2),:mydata(3),:mydata(4)
     FROM Sample.Person
     WHERE Home_State='MA' )
IF SQLCODE=0 {
   FOR i=1:1:15 {
     IF $DATA(mydata(i)) {
     WRITE "field ",i," = ",mydata(i),! }
   } }
 ELSE {WRITE "SQLCODE=",SQLCODE,! }
```

For further examples refer to Host Variable List Examples, below.

Using a Host Variable Array

A host variable array uses a single subscripted variable to contain all of the selected field values. This array is populated according to the order of field definition in the table, not the order of fields in the *select-item* list.

The following rules apply when using a host variable array in the **INTO** clause:

- The fields specified in the *select-item* list are selected into subscripts of a single host variable. Therefore, you do not have to match the number of items in the *select-item* list with the host variable count.

- The host variable subscripts are populated by the corresponding field position in the table definition. For example, the 6th field, as defined in the table definition, corresponds to mydata(6). All subscripts that do not correspond to a specified *select-item* remain undefined. The order of the items in the *select-item* has no effect on how subscripts are populated.

- A host variable array can only return field values from a single table.

- A host variable array can only return field values. It cannot return an aggregate value (such as a count, sum, or average), a function value, or a %CLASSNAME or %TABLENAME value. (You can return these by specifying a host variable argument that combines host variable list items with the host variable array.)

The following example selects four fields into a host variable array:

ObjectScript

```
&sql(SELECT %ID,Home_City,Name,SSN
     INTO :mydata()
     FROM Sample.Person
     WHERE Home_State='MA' )
 IF SQLCODE=0 {
   FOR i=0:1:15 {
     IF $DATA(mydata(i)) {
     WRITE "field ",i," = ",mydata(i),! }
   } }
 ELSE {WRITE "SQLCODE=",SQLCODE,! }
```

For further examples refer to Host Variable Array Examples, below.

For further details, refer to Host Variable as a Subscripted Array.

Arguments

:hostvar1

An output host variable that has been declared in the host language. When specified in an **INTO** clause, the variable name is preceded by a colon (:). A host variable can be a local variable (unsubscripted or subscripted) or an object property. You can specify multiple variables as a comma-separated list, as a single subscripted array variable, or a combination of a comma-separated list and a single subscripted array variable.

Host Variable Returning Field Values

The following Embedded SQL example selects three fields from the first record in the table (Embedded SQL always retrieves a single record), and uses **INTO** to set three corresponding unsubscripted host variables. These variables are then used by the ObjectScript **WRITE** commands. It is considered good program practice to immediately test the SQLCODE variable upon returning from Embedded SQL. If SQLCODE is not equal to 0, the values of output host variables are initialized to the empty string.

ObjectScript

```
WRITE !,"Going to get the first record"
&sql(SELECT Home_State, Name, Age
     INTO :state, :name, :age
     FROM Sample.Person)
IF SQLCODE=0 {
  WRITE !,"  Name=",name
  WRITE !,"  Age=",age
  WRITE !,"  Home State=",state }
ELSE {
  WRITE !,"SQL error ",SQLCODE  }
```

The following Embedded SQL example returns field values from a row resulting from the join of two tables. You must use a host variable list when returning fields from more than one table:

ObjectScript

```
&sql(SELECT P.Name,E.Title,E.Name,P.%TABLENAME,E.%TABLENAME
     INTO :name(1),:title,:name(2),:ptname,:etname
     FROM Sample.Person AS P LEFT JOIN
          Sample.Employee AS E ON E.Name %STARTSWITH 'B'
     WHERE P.Name %STARTSWITH 'A')
IF SQLCODE=0 {
     WRITE ptname," = ",name(1),!
     WRITE etname," = ",title,!
     WRITE etname," = ",name(2) }
ELSE {
     WRITE !,"SQL error ",SQLCODE  }
```

For restrictions on the use of input and output host variable values, refer to Host Variables.

Host Variables Returning Literal and Aggregate Values

Because output host variables are only valid when SQLCODE=0, it is important to avoid using the results of a query that issues an SQLCODE=100 (query returns no table data). SQLCODE=100 defaults all output host variables to the empty string, including returned literals and COUNT aggregates.

The following Embedded SQL example passes a host variable (*today*) into the SELECT statement, where a calculation results in the INTO clause variable value (*:tomorrow*). This host variable is passed out to the containing program. This query does not reference table fields and therefore does not specify a FROM clause. An Embedded SQL query without a FROM clause cannot issue SQLCODE=100. An Embedded SQL query with a FROM clause can issue SQLCODE=100, which would define all output variables to default null string values, including those such as *:tomorrow* that are not table field values.

ObjectScript

```
SET today=$HOROLOG
&sql(SELECT :today+1
     INTO :tomorrow )
IF SQLCODE=0 {
     WRITE !,"Tomorrow is: ",$ZDATE(tomorrow) }
ELSE {
     WRITE !,"SQL error ",SQLCODE  }
```

The following Embedded SQL example returns aggregate values. It uses the **COUNT** aggregate function to count the records in a table and **AVG** to average the Salary field values. The **INTO** clause returns these values to ObjectScript as two subscripted host variables.

Because both select-items are aggregates, this program always issues SQLCODE=0, even when the specified table contains no data. In this case, COUNT(*)=0 and AVG(Salary) is the default empty string.

ObjectScript

```
WRITE !,"Counting the records"
&sql(SELECT COUNT(*),AVG(Salary)
     INTO :agg(1),:agg(2)
     FROM Sample.Employee)
IF SQLCODE=0 {
     WRITE !,"Total Employee records= ",agg(1)
     WRITE !,"Average Employee salary= ",agg(2) }
ELSEIF SQLCODE=100 {
     WRITE !,"Total Employee records= ",agg(1) }
ELSE {
     WRITE !,"SQL error ",SQLCODE  }
```

The following Embedded SQL example is the same as the previous example, except it also returns a field value. Because the select-items includes a field value, this program can issue SQLCODE=100 when the specified table contains no data. In this example, if SQLCODE=100, COUNT(*) is the default empty string, not 0:

ObjectScript

```
WRITE !,"Counting the records"
&sql(SELECT COUNT(*),AVG(Salary),Salary
     INTO :agg(1),:agg(2),:pay
     FROM Sample.Employee)
IF SQLCODE=0 {
     WRITE !,"Total Employee records= ",agg(1)
     WRITE !,"Average Employee salary= ",agg(2)
     WRITE !,"Sample Employee salary=",pay }
ELSE {
     WRITE !,"SQL error ",SQLCODE  }
```

For restrictions on the use of input and output host variable values, refer to Host Variables.

Host Variable Array

The following two Embedded SQL examples use a host variable array to return the non-hidden data field values from a row. In these examples %ID is specified in the *select-item* list, because, by default, SELECT * does not return the RowId (though it does for Sample.Person); the RowId is always field 1. Note in Sample.Person, fields 4 and 9 can take NULL, field 5 is not a data field (it references Sample.Address), and field 10 is hidden.

The first example returns a specified number of fields (*firstflds*); hidden and non-data fields are included in this count, though not displayed. Using *firstflds* would be appropriate when returning a row from a table with many fields. Note that this example can return Field 0, which is the parent reference. Sample.Person is not a child table, so tflds(0) is undefined:

ObjectScript

```
&sql(SELECT *,%ID INTO :tflds()
       FROM Sample.Person )
 IF SQLCODE=0 {
    SET firstflds=14
    FOR i=0:1:firstflds {
       IF $DATA(tflds(i)) {
       WRITE "field ",i," = ",tflds(i),! }
    } }
 ELSE {WRITE "SQLCODE error=",SQLCODE,! }
```

The second example returns all the non-hidden data fields in Sample.Person. Note that this example does not attempt to return Field 0, the parent reference, because in Sample.Person tflds(0) is undefined, and would therefore generate an <UNDEFINED> error:

ObjectScript

```
&sql(SELECT *,%ID INTO :tflds()
       FROM Sample.Person )
IF SQLCODE=0 {
     SET x=1
     WHILE x '="" {
     WRITE "field ",x," = ",tflds(x),!
     SET x=$ORDER(tflds(x)) }
     }
ELSE { WRITE "SQLCODE error=",SQLCODE,! }
```

The following Embedded SQL example combines a comma-separated host variable list (for non-field values) and a host variable array (for field values):

ObjectScript

```
&sql(SELECT %TABLENAME,Name,Age,AVG(Age)
       INTO :tname,:tflds(),:ageavg
       FROM Sample.Person
       WHERE Age > 50 )
 IF SQLCODE=0 {
    WRITE "Table name is = ",tname,!
    FOR i=0:1:25 {
       IF $DATA(tflds(i)) {
       WRITE "field ",i," = ",tflds(i),! }
    }
       WRITE "Average age is = ",ageavg,! }
 ELSE {WRITE "SQLCODE=",SQLCODE,! }
```

See Also

- SELECT, DECLARE, FETCH statements

- VALUES clause

- Host Variables

- ObjectScript: SET command

ORDER BY (SQL)

A SELECT clause that specifies the sorting of rows in a result set.

Synopsis

```
ORDER BY orderItem

ORDER BY orderItem [ASC | DESC]

ORDER BY orderItem [ASC | DESC], orderItem2 [ASC | DESC]
```

Description

ORDER BY sorts the rows of a query result set by one or more specified ordering items, typically columns. Specify **ORDER BY** as the last clause in a SELECT statement, after the FROM, WHERE, GROUP BY, and HAVING clauses. For example:

SQL

```
SELECT Name, AVG(Age) AS AvgAge, Home_State
FROM Sample.Person
GROUP BY Home_State
ORDER BY AvgAge
```

- **ORDER BY** *orderItem* sorts the rows of a query result set by the values of the specified order item, such as a column. The rows are returned in ascending order.

  This statement returns the queried rows sorted by the `Home_State` column in ascending order.

  SQL

  ```
  SELECT Name,Age,Home_State
  FROM Sample.Person
  ORDER BY Home_State
  ```

  Example: Sort By Column Name

- **ORDER BY** *orderItem* **[ASC | DESC]** sorts the values in either ascending order (ASC) or descending order (DESC).

  This statement returns the queried rows sorted by the `Home_State` column in descending order.

  SQL

  ```
  SELECT Name,Age,Home_State
  FROM Sample.Person
  ORDER BY Home_State DESC
  ```

  Example: Using a TOP Clause with ORDER BY

- **ORDER BY** *orderItem* **[ASC | DESC],** *orderItem2* **[ASC | DESC]** sorts the values sequentially by one or more order items.

  This statement returns the queried rows sorted first by the `Home_State` column in ascending order, then sorted by the `Age` column in descending order.

  SQL

  ```
  SELECT Name,Age,Home_State
  FROM Sample.Person
  ORDER BY Home_State,Age DESC
  ```

  Examples:

  – Sort By Column Name

- Sort By Column Alias

- Sort By Column Number

The **ORDER BY** clause is applied after the execution of window functions in the **SELECT** list (including a window function's own **ORDER BY** clause). Therefore, the values returned by a window function are not affected by the **SELECT** query's **ORDER BY** clause.

If you omit the **ORDER BY** clause, the returned row order is unspecified and can differ with each statement execution.

ORDER BY sorts rows by the Logical (internal storage) data value, regardless of the current Select Mode setting. For more details on how **ORDER BY** sorts rows, see ORDER BY Collation.

Arguments

orderItem

An item, or comma-separated list of items, that specifies the order by which to sort the query result set. You can specify the items in *orderItem* as any combination of the following:

- The name of a column in the table. The column does not need to be specified in the **SELECT** list. Column names are not case-sensitive. Specifying a column name that is not in the table generates an SQLCODE -29 error.

- The alias of a column in the table. The alias must be specified in the **SELECT** list. Column aliases are not case-sensitive.

- The number of a column in the table, specified as an unsigned numeric literal. The number is based on the order of columns specified in the **SELECT** list. Specifying a column number that does not correspond to a **SELECT** list column results in an SQLCODE -5 error.

  If the first character of an *orderItem* is a number, InterSystems IRIS® assumes you are specifying a column number. Integer truncation rules apply to resolve a non-integer value to an integer. For example, 1.99 resolves to 1. Otherwise, it assumes *orderItem* is a column name or column alias.

  You cannot specify a column number as a variable or as the result of an expression. You cannot enclose a column number in parentheses.

- An expression evaluated on a column in the table, such as ORDER BY LENGTH(Name).

- A window function, such as ORDER BY ROW_NUMBER() OVER (PARTITION BY State).

- An aggregate function, provided that it is also specified in the **SELECT** list. Specifying an aggregate function only in the **ORDER BY** clause generates an SQLCODE -73 error.

With few exceptions, an *orderItem* must be specified as a literal. You cannot use a variable or other expression that provides a column name as a string. If an *orderItem* cannot be parsed as either a valid identifier (column name or column alias) or unsigned integer (column number), that *orderItem* is ignored and **ORDER BY** execution proceeds to the next *orderItem* in the comma-separated list. Some examples of ignored *orderItem* values include:

- Dynamic SQL ? input parameters

- Embedded SQL :var host variables

- Subqueries

- Expressions that resolve to a number

- Signed numbers

- Numbers enclosed in parentheses

If the *orderItem* property is a very long string or if you are attempting to use ORDER BY on multiple longer *orderItems*, InterSystems SQL will raise an error. To avoid this, you can define TRUNCATE collation on the relevant fields. In general, truncating at 128 characters is safe.

Examples

Sort By Column Name

This statement sorts by column names:

SQL

```
SELECT Name,Home_State,DOB
FROM Sample.Person
ORDER BY Home_State,Name
```

You can sort by column name whether or not the sort column is in the **SELECT** list. For example, this statement returns the same rows in the same order as the previous statement, even though its **SELECT** list does not include the Home_State column:

SQL

```
SELECT Name,DOB
FROM Sample.Person
ORDER BY Home_State,Name
```

You can sort by the RowID value even if the RowID is private and not listed in the **SELECT** list. Specify the %ID pseudo-column name as the *orderItem*, rather than the actual RowID name. For example:

SQL

```
SELECT Name,DOB
FROM Sample.Person
ORDER BY %ID
```

An **ORDER BY** clause can specify a table name or table alias as part of the *orderItem*:

SQL

```
SELECT P.Name AS People,E.Name As Employees
FROM Sample.Person AS P,Sample.Employee AS E
ORDER BY P.Name
```

An **ORDER BY** clause can use the arrow syntax (–>) operator to specify a column in a table that is not the base table:

SQL

```
SELECT Name,Company->Name AS CompName
FROM Sample.Employee ORDER BY Company->Name,Name
```

Sort By Column Alias

This statement sorts by column alias.

SQL

```
SELECT Name,Home_State AS HS,DOB
FROM Sample.Person
ORDER BY HS,Name
```

This statement sorts by an expression in the **SELECT** list that has an alias.

SQL

```
SELECT Name,Age,$PIECE(AVG(Age)-Age,'.',1) AS AgeDev
FROM Sample.Employee ORDER BY AgeDev,Name
```

Sort By Column Number

This statement sorts by column number, that is, the numeric sequence of the retrieved columns specified in the **SELECT** list. It sorts by Home_State (column 2), then by Name (column 1)

SQL

```
SELECT Name,Home_State,DOB
FROM Sample.Person
ORDER BY 2,1
```

This statement sorts by an expression in the **SELECT** list using a column number.

SQL

```
SELECT Name,Age,$PIECE(AVG(Age)-Age,'.',1)
FROM Sample.Employee ORDER BY 3,Name
```

When you sort **SELECT *** results by column number, if the RowID is public (default), it counts as column 1.

SQL

```
SELECT * FROM Sample.Person ORDER BY 1
```

Using a TOP Clause with ORDER BY

If a **SELECT** statement specifies an **ORDER BY** and a TOP clause, the returned "top" rows are based on the order specified in the **ORDER BY** clause. For example, this statement returns the 5 rows from MyTable that have the highest age value, ordered from older to younger.

SQL

```
SELECT TOP 5 Name,Age FROM MyTable ORDER BY Age DESC
```

Sorting by RowID changes which rows are selected by the **TOP** clause. For example, consider a table that has 100 rows with sequential RowIDs. These statements returns rows 1, 2, 3, 4, 5 and rows 100, 99, 98, 97, 96, respectively.

SQL

```
SELECT TOP 5 %ID FROM MyTable ORDER BY %ID
```

SQL

```
SELECT TOP 5 %ID FROM MyTable ORDER BY %ID DESC
```

Sort Based on List Data

This statement sorts by a column containing InterSystems IRIS list data. Because an InterSystems IRIS list is an encoded character string that begins with formatting characters, this statement uses **$LISTTOSTRING** to sort by the actual column value, rather than the list element encoding:

SQL

```
SELECT Name,FavoriteColors
FROM Sample.Person
WHERE FavoriteColors IS NOT NULL
ORDER BY $LISTTOSTRING(FavoriteColors)
```

Sort Items Based on Host Variable Values

You can use the CASE expression to define a general-purpose query that can be ordered based on a supplied host variable value. For example, this statement can order by either `Name` or `Age`, depending on the value of *var*:

SQL

```
SELECT Name,Age FROM Sample.Person ORDER BY
CASE WHEN :var=1 then Name
     WHEN :var=2 then Age END
```

This statement specifies two **CASE** expressions. It orders by whichever case evaluates to true. If both cases evaluate to true, it orders by `Country`, and within `Country` by `City`:

```
SELECT Country,City FROM Sample.Person ORDER BY
CASE WHEN :var1=1 then Country END,
     WHEN :var2=1 then City END
```

Specify the ASC and DESC argument after the **CASE** END keyword.

You must specify fields in a **CASE** expression by column name. Column aliases and column numbers are not permitted in this context.

Sort Items Using Dynamic SQL and Embedded SQL

Dynamic SQL can use an input parameter to supply a literal value to an **ORDER BY** clause. It cannot use an input parameter to supply a column name, column alias, column number, or collation keyword. This Dynamic SQL example uses an input parameter to sort result set rows by first name:

ObjectScript

```
set myquery = 4
set myquery(1) = "SELECT TOP ? Name,Age,"
set myquery(2) = "CURRENT_DATE AS Today"
set myquery(3) = "FROM Sample.Person WHERE Age > ?"
set myquery(4) = "ORDER BY $PIECE(Name,',',?)"

set tStatement = ##class(%SQL.Statement).%New()
set qStatus = tStatement.%Prepare(.myquery)
if qStatus '= 1 {
  write "%Prepare failed:"
  do $System.Status.DisplayError(qStatus)
  quit }

set rset = tStatement.%Execute(10,60,2)
do rset.%Display()
write !,"%Display SQLCODE=",rset.%SQLCODE
```

This cursor-based Embedded SQL example performs the same operation:

ObjectScript

```
SET topnum=10,agemin=60,firstname=2

&sql(DECLARE pCursor CURSOR FOR
     SELECT TOP :topnum Name,Age,CURRENT_DATE AS Today
     INTO :name,:years,:today FROM Sample.Person
     WHERE Age > :agemin
     ORDER BY $PIECE(Name,',',:firstname) )

&sql(OPEN pCursor)
     QUIT:(SQLCODE'=0)

FOR { &sql(FETCH pCursor)
     QUIT:SQLCODE
     WRITE "Name=",name," Age=",years," today=",today,!
     }

&sql(CLOSE pCursor)
```

Limitations

- If your **SELECT** query specifies an **ORDER BY** clause, the resulting data is not updateable. Thus, if you specify a subsequent **DECLARE CURSOR FOR UPDATE** statement, the **FOR UPDATE** clause is ignored, and the cursor is declared read-only.

- If the **ORDER BY** applies to a UNION, an ordering item must be a number or a simple column name. It cannot be an expression. If a column name is used, it refers to result columns as they are named in the first **SELECT** list of the **UNION**.

- When used in a subquery, an **ORDER BY** clause must be paired with a TOP clause. This may be a **TOP ALL** clause. For example, this query is not valid because it uses a **DISTINCT** clause in the sorted subquery:

```
SELECT Name FROM Sample.Person WHERE Name =
   (SELECT DISTINCT Name FROM Sample.Employee ORDER BY Title)
```

The query is valid because it uses **TOP ALL** in the sorted subquery instead:

SQL

```
SELECT Name FROM Sample.Person WHERE Name =
   (SELECT TOP ALL Name FROM Sample.Employee ORDER BY Title)
```

- Running a query with an **ORDER BY** *orderItem* value that exceeds 400 characters can result in an SQL -400 fatal error. This occurs because of a limitation in the maximum encoded length of a global reference, which is a fixed InterSystems IRIS system limit. To prevent this problem, use a truncation length in the collation setting for the field that is the basis of the **ORDER BY** clause.

For example, suppose the `NarrativeSummary` column of this query exceeds 400 characters:

SQL

```
SELECT LocationCity,NarrativeSummary FROM Aviation.Event
WHERE LocationCity %STARTSWITH 'Be'
ORDER BY NarrativeSummary
```

Adding a collation function with a *maxlen* truncation length allows this query to execute successfully.

SQL

```
SELECT LocationCity,NarrativeSummary FROM Aviation.Event
WHERE LocationCity %STARTSWITH 'Be'
ORDER BY %SQLUPPER(NarrativeSummary,400)
```

Performance

Each literal value used in an **ORDER BY** clause generates a different cached query. Literal substitution is not performed on **ORDER BY** literals. This is because **ORDER BY** can use an integer to specify a column number. Changing this integer would result in a fundamentally different query.

More About

ORDER BY Collation

Sorting is done in collation order. By default, the ordering of string values is done based on the collation specified for the **ORDER BY** *orderItem* column when it was created. If your InterSystems IRIS namespace uses the default string collation of SQLUPPER, then **ORDER BY** collation is not case-sensitive.

Ordering of numeric data type fields is done based on numeric collation. For expressions, the default collation is EXACT.

You can override the default collation for a column by applying a collation function. For example: `ORDER BY %EXACT(Name)`. You cannot apply a collation function to a [column alias](). Attempting to do so generates an SQLCODE -29 error.

The default ascending collation sequence considers NULL to be the lowest value, followed by the empty string (''). **ORDER BY** does not distinguish between the empty string and strings that consist only of blank spaces.

If the collation specified for a column is alphanumeric, leading numbers are sorted in character collation sequence, not integer sequence. To order in integer sequence, you can use the [%PLUS]() collation function, but this function treats all non-numeric characters as 0.

To properly sort mixed numeric strings in numeric sequence, you must specify more than one **ORDER BY** *orderItem*. Consider a `Home_Street` column that has this format:

Number StreetName StreetType

`Number` is an integer house number. `StreetName` and `StreetType` are strings that combine to form the full street name, such as `"Elm Street"`.

This statement sorts street addresses in character collation sequence.

SQL

```
SELECT Name,Home_Street FROM Sample.Person
ORDER BY Home_Street
```

This statement sorts the house number in integer sequence and the street name in character collation sequence. This statement contains an expression and works only with a column name, not a column alias or column number.

SQL

```
SELECT Name,Home_Street FROM Sample.Person
ORDER BY $PIECE(%PLUS(Home_Street),' ',1),$PIECE(Home_Street,' ',2),$PIECE(Home_Street,' ',3)
```

ASC and DESC Collation

Sorting can be specified for each column in ascending or descending collation sequence order, as specified by the optional ASC (ascending) or DESC (descending) keyword following the column identifier. If ASC or DESC is not specified, **ORDER BY** sorts that column in ascending order. You cannot specify the ASC or DESC keyword using a Dynamic SQL ? input parameter or an Embedded SQL `:var` host variable.

NULL is always the lowest value in ASC sequence and the highest value in DESC sequence.

Multiple comma-separated **ORDER BY** values specify a hierarchy of sort operations. For example, this statement sorts the data values of the third-listed item (C) in the **SELECT** clause list in ascending order; within this sequence, it sorts the seventh-listed item (J) values in descending order; within this, it sorts the first-listed item (A) values in ascending order.

SQL

```
SELECT A,B,C,M,E,X,J
FROM LetterTable
ORDER BY 3,7 DESC,1 ASC
```

Duplicate columns in the list of **ORDER BY** values have no effect. This is because the second sort is within the order of the first sort. For example, `ORDER BY Name ASC, Name DESC` sorts the Name column in ascending order.

NLS Collation

If you have specified a non-default NLS (National Language Support) collation, you must make sure that all collations are aligned and use the exact same national collation sequence. This includes not only globals used by the tables, but also globals used for indexes, in temporary files such as in IRISTEMP and process-private globals. For more details, see [SQL Collation and NLS Collations]()

See Also

- SELECT
- UNION
- TOP
- Collation
- Querying the Database
- SQLCODE error messages

TOP (SQL)

A SELECT clause that specifies how many rows to return.

Synopsis

```
SELECT  [DISTINCT clause]
    [TOP {[((]int[))] | ALL}]
    select-item{,select-item}
```

Arguments

Argument	Description
int	Limits the number of rows returned to the specified integer number. The *int* argument can be either a positive integer, a Dynamic SQL input parameter (?) or an Embedded SQL host variable (:var) that resolve to a positive integer.
	In Dynamic SQL, the *int* value can optionally be enclosed with single parentheses or double parentheses (double parentheses are the preferred syntax); these parentheses suppress literal substitution of the *int* value in the corresponding cached query.
ALL	TOP ALL is only meaningful in a subquery or in a CREATE VIEW statement. It is used to support the use of an ORDER BY clause in these situations, fulfilling the requirement that an ORDER BY clause must be paired with a TOP clause in a subquery or a query used in a CREATE VIEW. TOP ALL does not restrict the number of rows returned.

Description

The optional TOP clause appears after the **SELECT** keyword and the optional DISTINCT clause, and before the first *select-item*.

The TOP keyword is used in Dynamic SQL and in cursor-based Embedded SQL. In non-cursor Embedded SQL the only meaningful use of the TOP keyword is TOP 0. Any other TOP *int* (where *int* is any non-zero integer) is valid but not meaningful because a **SELECT** in non-cursor Embedded SQL always returns at most one row of data.

The TOP clause of a SELECT statement limits the number of rows returned to the number specified in *int*. If no TOP clause is specified, the default is to display all the rows that meet the **SELECT** criteria. If a TOP clause is specified, the number of rows displayed is either *int* or all of the rows that fulfill the query predicate requirements, whichever is smaller. If you specify ALL, **SELECT** returns all the rows in the table that fulfill the query predicate requirements.

If no ORDER BY clause is specified in the query, the records chosen to be returned as the "top" rows are unpredictable. If an ORDER BY clause is specified, the top rows accord to the order specified in that clause.

The DISTINCT clause (if specified) is applied before TOP, specifying that (at most) *int* number of unique values are to be returned.

TOP short circuits when all rows have been delivered. Thus, if you select until you get SQLCODE 100, the FETCH that sets SQLCODE 100 is instant.

When accessing data through a view, or through a FROM clause subquery, you can limit the number of rows returned by using the %vid view ID, rather than (or in addition to) the TOP clause. For further details on using %vid, refer to Defining and Using Views.

The TOP int Value

The *int* numeric value can be an integer, or a numeric string, a Dynamic SQL input parameter (?), or an input host variable (:var) that resolve to an integer value.

The *int* value specifies the number of rows to return. Permitted values are 0 and positive numbers. You cannot specify the *int* value as an arithmetic expression, field name, subquery column alias, scalar function, or aggregate function. A fractional number or a numeric string is parsed as its integer value. Zero (0) is a valid *int* value. TOP 0 executes the query but returns no data.

TOP ALL must be specified as a keyword in the query. You cannot specify ALL as a ? input parameter or :var host variable value. The query parser interprets the string "ALL" supplied in this way as a numeric string with a value of 0.

Note that the TOP argument metadata is returned as database driver data type 12 (VARCHAR) rather than 4 (INTEGER) because it is possible to specify TOP *int* as a numeric string or an integer.

TOP and Cached Queries

An *int* value can be specified with or without enclosing parentheses. These parentheses affect how a Dynamic SQL query is cached (non-cursor Embedded SQL queries are not cached). An *int* value without parentheses is converted to a ? parameter variable in the cached query. This means that repeatedly invoking the same query with different TOP *int* values invokes the same cached query, rather than preparing and optimizing the query each time.

Enclosing parentheses suppress literal substitution. For example, TOP ((7)). When *int* is enclosed in parentheses, the cached query preserves the specific *int* value. Re-invoking the query with the same TOP *int* value uses the cached query; invoking the query with a different TOP *int* value causes SQL to prepare, optimize, and cache this new version of the query.

TOP ALL is not cached as a ? parameter variable. ALL is parsed as a keyword, not a literal. Therefore, the same query with TOP 7 and with TOP ALL will generate two different cached queries.

An *int* value can be specified with or without enclosing parentheses. These parentheses affect how a Dynamic SQL query is cached (non-cursor Embedded SQL queries are not cached). An *int* value without parentheses is converted to a ? parameter variable in the cached query. This means that repeatedly invoking the same query with different TOP *int* values invokes the same cached query, rather than preparing and optimizing the query each time.

Enclosing parentheses suppress literal substitution. For example, TOP ((7)). When *int* is enclosed in parentheses, the cached query preserves the specific *int* value. Re-invoking the query with the same TOP *int* value uses the cached query; invoking the query with a different TOP *int* value causes SQL to prepare, optimize, and cache this new version of the query.

TOP ALL is not cached as a ? parameter variable. ALL is parsed as a keyword, not a literal. Therefore, the same query with TOP 7 and with TOP ALL will generate two different cached queries.

TOP and ORDER BY

TOP is generally used in a **SELECT** with an ORDER BY clause. Note that the default ascending ORDER BY collation sequence considers NULL to be the lowest ("top") value, followed by the empty string (").

TOP is required in a subquery **SELECT** or a CREATE VIEW **SELECT** when specifying an ORDER BY clause. In these cases you can specify either TOP *int* (to limit the number of rows to return) or TOP ALL.

TOP ALL is only used in a subquery or in a **CREATE VIEW** statement. It is used to support the use of an ORDER BY clause in these situations, fulfilling the requirement that an ORDER BY clause must be paired with a TOP clause in a subquery or a **CREATE VIEW** query. TOP ALL does not restrict the number of rows returned. TOP ALL ... ORDER BY *does not* change default **SELECT** optimization. The ALL keyword cannot be enclosed in parentheses.

TOP Optimization

By default, a **SELECT** optimizes for fastest time to return all data. Adding both a TOP *int* clause and an ORDER BY clause optimizes for fastest time to return first row. (Note that both clauses are required to change the optimization.) You can use the %SYS.PTools.StatsSQL class TotalTimeToFirstRow property to return the time required to return the first row.

The following are special case optimizations:

- You may wish to use the TOP and ORDER BY optimization strategy without limiting the number of rows returned; for example, if you are returning data that is displayed in page units. In such a case, you may want to issue a TOP clause with an *int* value larger than the total number of rows.

- You may wish to limit the number of rows returned and specify their order without changing the default **SELECT** optimization. In this case, specify a TOP clause, an ORDER BY clause, and the %NOTOPOPT keyword to preserve fastest time to return all data optimization. See FROM for more details.

TOP with Aggregates and Functions

An aggregate function or a scalar function can only return a single value. If the query *select-item* list contains *only* aggregates and functions, the application of the TOP clause is as follows:

- If the *select-item* list contains an aggregate function, for example COUNT(*) or AVG(Age), and does not contain any field references, no more than one row is returned, regardless of the TOP *int* value or the presence of an ORDER BY clause. These clauses are validated, but ignored. This is shown in the following examples:

  SQL

  ```
  SELECT TOP 5 AVG(Age),CURRENT_TIMESTAMP(3) FROM Sample.Person
    /* returns 1 row */
  ```

  SQL

  ```
  SELECT TOP 1 AVG(Age),CURRENT_TIMESTAMP(3) FROM Sample.Person ORDER BY Age
    /* returns 1 row */
  ```

- If the *select-item* list contains one or more scalar functions, expressions, literals (such as %TABLENAME), subqueries, or host variables, and does not contain any field references or aggregates, the TOP clause is applied. This is shown in the following example:

  SQL

  ```
  SELECT TOP 5 ROUND(678.987,2),CURRENT_TIMESTAMP(3) FROM Sample.Person
    /* returns 5 identical rows */
  ```

  The actual number of rows returned depends on the number of rows in the table, even when table fields are not referenced. For example:

  SQL

  ```
  SELECT TOP 300 CURRENT_TIMESTAMP(3) FROM Sample.Person
    /* returns either the number of rows in Sample.Person
       or 300 rows, whichever is smaller */
  ```

  When the query is restricted by a predicate condition, the number of rows returned is restricted by that condition, even when table fields are not referenced in the *select-item* list. For example:

  SQL

  ```
  SELECT TOP 300 CURRENT_TIMESTAMP(3) FROM Sample.Person WHERE Home_State = 'MA'
    /* returns either the number of rows in Sample.Person
       where Home_State = 'MA'
       or 300 rows, whichever is smaller */
  ```

- If the **SELECT** statement does not contain a FROM clause, at most one row is returned, regardless of the TOP value. For example:

  SQL

  ```
  SELECT TOP 5 ROUND(678.987,2),CURRENT_TIMESTAMP(3)
    /* returns 1 row */
  ```

- The DISTINCT clause further limits the TOP clause. If there are fewer distinct values than the TOP value, only the rows with distinct values are returned. When only scalar functions are referenced, only one row is returned. For example:

  SQL

  ```
  SELECT DISTINCT TOP 15 CURRENT_TIMESTAMP(3) FROM Sample.Person
    /* returns 1 row */
  ```

- TOP 0 always returns no rows, regardless of the contents of the *select-item* list, or whether the **SELECT** statement contains a FROM clause or a DISTINCT clause.

  In non-cursor Embedded SQL, a query with TOP 0 returns no rows and sets SQLCODE=100; a non-cursor Embedded SQL query with TOP 1 (or any other TOP *int* value) returns one row and sets SQLCODE=0. In cursor-based Embedded SQL, completion of the fetch loop always sets SQLCODE=100, regardless of the TOP *int* value.

Examples

The following query returns the first 20 rows retrieved from Sample.Person in the order that they are stored in the database. This record order is generally not predictable.

SQL

```
SELECT TOP 20 Home_State,Name FROM Sample.Person
```

The following query returns the first 20 distinct Home_State values retrieved from Sample.Person in ascending collation sequence order.

SQL

```
SELECT DISTINCT TOP 20 Home_State FROM Sample.Person ORDER BY Home_State
```

The following query returns the first 40 distinct FavoriteColor values. The "top" rows reflect the ORDER BY clause sequencing of all of the rows in Sample.Person in descending (DESC) collation sequence. Descending collation sequence is used rather than the default ascending collation sequence because the FavoriteColors field is known to have NULLs, which would appear at the top of the ascending collation sequence.

SQL

```
SELECT DISTINCT TOP 40 FavoriteColors FROM Sample.Person
    ORDER BY FavoriteColors DESC
```

Also note in the preceding example that because FavoriteColors is a list field, the collation sequence includes the element length byte. Thus six-letter elements (YELLOW, PURPLE, ORANGE) collate together, listed before five-letter elements (WHITE, GREEN, etc.).

Dynamic SQL can specify the *int* value as an input parameter (indicated by "?"). In the following example, the TOP ? input parameter is set to 10 by the **%Execute** method:

ObjectScript

```
SET myquery = "SELECT TOP ? Name,Age FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(10)
DO rset.%Display()
```

The following cursor-based Embedded SQL example performs the same operation:

ObjectScript

```
SET topnum=10
&sql(DECLARE pCursor CURSOR FOR
     SELECT TOP :topnum Name,Age INTO :name,:years FROM Sample.Person
     )
&sql(OPEN pCursor)
     QUIT:(SQLCODE'=0)
FOR { &sql(FETCH pCursor)
     QUIT:SQLCODE
     WRITE "Name=",name," Age=",years,!
     }
&sql(CLOSE pCursor)
```

See Also

- SELECT statement

- DISTINCT clause

- ORDER BY clause

- Querying the Database

UNION (SQL)

Combines two or more SELECT statements.

Synopsis

```
select-statement {UNION [ALL] [%PARALLEL] select-statement}

select-statement {UNION [ALL]  [%PARALLEL] (query)}

(query) {UNION [ALL]  [%PARALLEL] select-statement}

(query) {UNION [ALL]  [%PARALLEL] (query)}
```

Description

A **UNION** combines two or more queries into a single query that retrieves data into a result. The queries that are combined by a **UNION** can be simple queries, consisting of a single SELECT statement, or compound queries.

For a union to be possible between **SELECT** statements, the number of columns specified in each leg must match. Specifying **SELECT**s with different numbers of columns results in an SQLCODE -9 error. You can specify a NULL column in one **SELECT** to pair with a data column in another **SELECT** in order to match the number of columns. For example:

SQL

```
SELECT Name,Salary,BirthDate
FROM Sample.Employee
UNION ALL
SELECT Name,NULL,BirthDate
FROM Sample.Person
```

CAUTION: To use the **SELECT *** syntax in a **UNION**, the tables must contain the same number of columns. Therefore, future changes to the table definition by adding or deleting a column may cause unforeseen errors in unions of this sort.

InterSystems SQL determines the result column data types by automatically evaluating all legs of the **UNION** query and returning the data type with the highest precedence as follows: VARCHAR, DOUBLE, NUMERIC, BIGINT, INTEGER, SMALLINT, TINYINT. Other data types, such as DATE, are not assigned precedence. For example, the following program returns data type TINYINT, even though the DATE data type has a higher precedence in other contexts.

SQL

```
SELECT MyTinyIntField FROM Table1
     UNION ALL
SELECT MyDateField FROM Table2
```

If you want to return a data type other than the ones listed, you have to use an explicit **CAST** statement, as shown in the following example:

SQL

```
SELECT CAST(MyTinyInt AS DATE) FROM Table1
     UNION ALL
SELECT MyDateField FROM Table2
```

If the columns in the legs of the union differ in length, precision, or scale, the result column is assigned the largest value.

Result column names are taken from the name of the column (or column alias) in the first leg of the union. In situations where the corresponding columns in the two legs do not have the same names, it may be useful to use the same column alias in all of the legs to identify the result column.

If any column in any of the UNION legs is nullable, the result column metadata is reported as nullable.

String fields in the **UNION** result have the collation type of the corresponding **SELECT** fields, but are assigned EXACT collation if the field collations do not match.

UNION and UNION ALL

An ordinary **UNION** eliminates duplicate rows (all values identical) from the result. A **UNION ALL** preserves duplicate rows in the result.

Fields of different precision do not have identical values. For example, the values 33 (data type NUMERIC(9)) and 33.00 (data type NUMERIC(9,2)) are not considered identical.

Fields with different collations do not have identical values. For example, MyStringField and %SQLUPPER(MyStringField) are not considered identical, even if both values are all uppercase.

TOP and ORDER BY Clauses

A **UNION** statement can conclude with an ORDER BY clause which orders the result. This **ORDER BY** applies to the whole statement; it must be part of the outermost query, not a subquery. It does not have to be paired with a **TOP** clause. The following example shows this use of **ORDER BY**: the two **SELECT** statements select data, the data is combined by the **UNION**, then the **ORDER BY** sequences the results:

SQL

```
SELECT Name,Home_Zip FROM Sample.Person
  WHERE Home_Zip %STARTSWITH 9
UNION
SELECT Name,Office_Zip FROM Sample.Employee
  WHERE Office_Zip %STARTSWITH 8
ORDER BY Home_Zip
```

Using a column number in **ORDER BY** that does not correspond to a **SELECT** list column results in an SQLCODE -5 error. Using a column name in **ORDER BY** that does not correspond to a **SELECT** list column results in an SQLCODE -6 error.

Either **SELECT** statements (or both) in a union can also contain an ORDER BY clause, but it must be paired with a TOP clause. This **ORDER BY** is applied to determine which rows are selected by the **TOP** clause. The following example shows this use of **ORDER BY**: the two **SELECT** statements each use an **ORDER BY** to sequence their rows, which determines which rows are selected as the top rows. The selected data is combined by the **UNION**, then the final **ORDER BY** sequences the results:

SQL

```
SELECT TOP 5 Name,Home_Zip FROM Sample.Person
  WHERE Home_Zip %STARTSWITH 9
  ORDER BY Name
UNION
SELECT TOP 5 Name,Office_Zip FROM Sample.Employee
  WHERE Office_Zip %STARTSWITH 8
  ORDER BY Office_Zip
ORDER BY Home_Zip
```

TOP may apply to the first **SELECT** in the union, or to the result of the union, depending on the placement of the **ORDER BY** clause:

- TOP...ORDER BY applies to **UNION** result: the **UNION** is within a FROM clause subquery, and TOP and ORDER BY are applied to the results of the **UNION**. For example:

SQL

```
SELECT TOP 10 Name,Home_Zip
  FROM (SELECT Name,Home_Zip FROM Sample.Person
          WHERE Name %STARTSWITH 'A'
       UNION
       SELECT Name,Home_Zip FROM Sample.Person
          WHERE Home_Zip %STARTSWITH 8)
ORDER BY Home_Zip
```

- TOP applies to first **SELECT**; ORDER BY applies to **UNION** result. For example:

SQL

```
SELECT TOP 10 Name,Home_Zip
  FROM Sample.Person
  WHERE Name %STARTSWITH 'A'
UNION
SELECT Name,Home_Zip FROM Sample.Person
  WHERE Home_Zip %STARTSWITH 8
ORDER BY Home_Zip
```

Enclosing Parentheses

UNION supports optional enclosing parentheses for either or both of its **SELECT** statements, or for the entire **UNION** statement. You may specify one or more pairs of enclosing parentheses. The following are all valid uses of enclosing parentheses:

```
(SELECT ...) UNION SELECT ...
(SELECT ...) UNION (SELECT ...)
((SELECT ...)) UNION ((SELECT ...))
(SELECT ... UNION SELECT ...)
((SELECT ...) UNION (SELECT ...))
```

Each use of parentheses generates a separate cached query.

UNION/OR Optimization

By default, SQL automatic optimization transforms UNION subqueries to OR conditions, where deemed appropriate. This UNION/OR transformation allows EXISTS and other low-level predicates to migrate to top-level conditions where they are available to InterSystems IRIS query optimizer indexing. This default transformation is desirable in most situations. However, in some situations this UNION/OR transformation imposes a significant overhead burden. The %NOUNIONOROPT query optimization option disables this automatic UNION/OR transformation for all conditions in the WHERE clause associated with the FROM clause. Thus, in a complex query, you can disable automatic UNION/OR optimization for one subquery while allowing it in other subqueries. For further information on %NOUNIONOROPT, refer to the FROM clause.

If a condition involving a subquery is applied to a UNION, it is applied within each union operand, rather than at the end. This allows subquery optimizations to be applied in each UNION operand. For descriptions of subquery optimization options, refer to the FROM clause. In the following example, the WHERE clause condition is applied to each of the subqueries in the union, rather than to the result of the union:

SQL

```
SELECT Name,Age FROM
  (SELECT Name,Age FROM Sample.Person
   UNION SELECT Name,Age FROM Sample.Employee)
WHERE Age IN (SELECT TOP 5 Age FROM Sample.Employee WHERE Age>55 ORDER BY Age)
```

UNION ALL Aggregate Optimization

SQL automatic optimization of a UNION ALL pushes a top-level aggregate into the legs of the union. This can result in significantly improved performance with or without the %PARALLEL keyword, For example:

SQL

```
SELECT COUNT(*) FROM (SELECT item1 FROM table1 UNION ALL SELECT item2 FROM table2)
```

is optimized as:

SQL

```
SELECT SUM(y) FROM (SELECT COUNT(*) AS y FROM table1 UNION ALL SELECT COUNT(*) AS y FROM table2)
```

This optimization applies to all top-level aggregate functions (not just COUNT), including queries with multiple top-level aggregate functions. For this optimization to be applied, the outer query must be a "onerow" query, with no WHERE or GROUP BY clause, it cannot reference %VID, and the UNION ALL must be the only stream in its FROM clause. The aggregates cannot be nested, and any aggregate function used cannot use %FOREACH() grouping or DISTINCT.

Parallel Processing

The %PARALLEL keyword supports parallelism and distributed processing on a multiprocessor system. It causes InterSystems IRIS to perform parallel processing on the UNION queries, assigning each query to a separate process on the same machine. In some cases that process will send the query to a different machine to be processed. These processes communicate via pipes, with InterSystems IRIS creating one or more temporary files to hold subquery results. The main process combines the resulting rows and returns the final results. For further details, refer to the Show Plan for a UNION query, comparing the Show Plan with and without the %PARALLEL keyword. To determine the number of processors on the current system use the **%SYSTEM.Util.NumberOfCPUs()** method.

In general, the more effort expended to produce each row, the more beneficial %PARALLEL becomes.

Specifying the %PARALLEL keyword disables automatic UNION-to-OR optimizations.

The following examples show the use of the %PARALLEL keyword:

SQL

```
SELECT Name FROM Sample.Employee WHERE Name %STARTSWITH 'A'
UNION %PARALLEL
SELECT Name FROM Sample.Person WHERE Name %STARTSWITH 'A'
ORDER BY Name
```

SQL

```
SELECT Name FROM Sample.Employee WHERE Name %STARTSWITH 'A'
UNION ALL %PARALLEL
SELECT Name FROM Sample.Person WHERE Name %STARTSWITH 'A'
ORDER BY Name
```

%PARALLEL is intended for **SELECT** queries and their subqueries. An INSERT command subquery cannot use %PARALLEL.

Adding the %PARALLEL keyword may not be appropriate for all UNION queries, and may result in an error. The following SQL constructs generally do not support UNION %PARALLEL execution: an outer join, a correlated field, an IN predicate condition containing a subquery, or a collection predicate. UNION %PARALLEL is supported for a FOR SOME predicate, but not for a FOR SOME %ELEMENT collection predicate. To determine if a UNION query can successfully use %PARALLEL, test each leg of the UNION separately. Separately test each leg query by adding a FROM %PARALLEL keyword. If one of the FROM %PARALLEL queries generates a query plan that does not show parallelization, then the UNION query will not support %PARALLEL.

UNION ALL and Aggregate Functions

SQL automatic optimization pushes UNION ALL aggregate functions into the union leg subqueries. SQL calculates the aggregate value for each subquery, and then combines the results to return the original aggregate value. For example:

SQL

```
SELECT COUNT(Name) FROM (SELECT Name FROM Sample.Person
                          UNION ALL SELECT Name FROM Sample.Employee)
```

Is optimized as:

SQL

```
SELECT SUM(y) FROM (SELECT COUNT(Name) AS y FROM Sample.Person
                      UNION ALL SELECT COUNT(Name) AS y FROM Sample.Employee)
```

This can result in substantial performance improvement. This optimization is applied with or without the %PARALLEL keyword. This optimization is applied to multiple aggregate functions.

This optimization transform only occurs under the following circumstances:

- The outer query FROM clause must contain only a UNION ALL statement.

- The outer query cannot contain a WHERE clause or a GROUP BY clause.

- The outer query cannot contain a %VID (view ID) field.

- Aggregate functions cannot contain a DISTINCT or %FOREACH keyword.

- Aggregate functions cannot be nested.

Arguments

ALL

An optional keyword literal. If specified, duplicate data values are returned. If omitted, duplicate data values are suppressed.

%PARALLEL

An optional argument that specifies the %PARALLEL keyword. If specified, each side of the union is run in parallel as a separate process.

select-statement

A SELECT statement, which retrieves data from a database.

query

A query that combines one or more **SELECT** statements.

Examples

The following example creates a result that contains a row for every Name found in each of the two tables; if a Name is found in both tables, two rows are created. When the Name is an employee, it lists the office location, concatenated with the word "office" as State, and the employee's Title. When Name is a person, it lists the home location, concatenated with the word "home" as State, and <null> for Title. The ORDER BY clause operates on the result; the combined rows are ordered by Name:

SQL

```
SELECT Name,Office_State||' office' AS State,Title
FROM Sample.Employee
UNION
SELECT Name,Home_State||' home',NULL
FROM Sample.Person
ORDER BY Name
```

The following two examples show the effects of the ALL keyword. In the first example, **UNION** returns only unique values. In the second example, **UNION ALL** returns all values, including duplicates:

SQL

```
SELECT Name
FROM Sample.Employee
WHERE Name %STARTSWITH 'A'
UNION
SELECT Name
FROM Sample.Person
WHERE Name %STARTSWITH 'A'
ORDER BY Name
```

SQL

```
SELECT Name
FROM Sample.Employee
WHERE Name %STARTSWITH 'A'
UNION ALL
SELECT Name
FROM Sample.Person
WHERE Name %STARTSWITH 'A'
ORDER BY Name
```

See Also

- SELECT

- ORDER BY clause, TOP clause

- CREATE QUERY, CREATE PROCEDURE

- Querying the Database

- SQLCODE error messages

VALUES (SQL)

An INSERT/UPDATE clause that specifies data values for use in fields.

Synopsis

```
(field1{,fieldn})
     VALUES (value1{,valuen})
```

Description

The **VALUES** clause is used in an INSERT, UPDATE, or INSERT OR UPDATE statement to specify the data values to insert into the fields. Typically:

- **INSERT** queries use the following syntax:

```
INSERT INTO tablename (fieldname1,fieldname2,...)
     VALUES (value1,value2,...)
```

- **UPDATE** queries use the following syntax:

```
UPDATE tablename (fieldname1,fieldname2,...)
     VALUES (value1,value2,...)
```

The elements in the **VALUES** clause correspond in sequence to the fields specified after the table name. Note if there is only one value element specified in the **VALUES** clause, it is not necessary to enclose the element in parentheses.

The following example shows an **INSERT** statement that adds a single row to the "Employee" table:

SQL

```
INSERT INTO Employee (Name,SocSec,Telephone)
VALUES("Boswell",333448888,"546-7989")
```

SQL

```
INSERT INTO Employee (Name,SocSec,Telephone)
VALUES ('Boswell',333448888,'546-7989')
```

Insert and update queries can use a **VALUES** clause without requiring you to explicitly specify a list of field names after the table name. In order to omit the list of field names after the table name, your query must meet the following two criteria:

- The number of values specified in the **VALUES** clause is the same as the number of fields in the table (exclusive of the ID field).

- The values in the **VALUES** clause are listed in order of the internal column numbers of the fields, beginning with column 2. Column 1 is always reserved for the system-generated ID field, and is not specified in a **VALUES** clause.

For example, the query:

SQL

```
INSERT INTO Sample.Person VALUES (5,'John')
```

is equivalent to the query:

SQL

```
INSERT INTO Sample.Person (Age,Name) VALUES (5,'John')
```

if the table "Sample.Person" has exactly two user-defined fields.

In this example, the value 5 is assigned to the field with the lower column number, and the value "John" is assigned to the other field.

A **VALUES** clause can specify an element of an array, as in the following embedded SQL example:

ObjectScript

```
&sql( UPDATE Person(Tel)
      VALUES :per('tel',)
      WHERE ID = :id )
```

An **UPDATE** query can also reference an array with unspecified last subscript. Whereas **INSERT** uses the presence and absence of array elements to assign values and default values to a newly created row, **UPDATE** uses the presence of an array element to indicate that the corresponding field should be updated. For example, consider the following array for a table with six columns:

```
emp("profile",2)="Smith"
emp("profile",3)=2
emp("profile",3,1)="1441 Main St."
emp("profile",3,2)="Cableton, IL 60433"
emp("profile",5)=NULL
emp("profile",7)=25
emp("profile","next")="F"
```

Column 1 is always reserved for the ID field, and is not user-specified. The inserted "Employee" row has Column 2 ("Name") set to "Smith"; Column 3 ("Address") set to a two-line value; Column 4 ("Department") is not specified, and is thus set to the default; and Column 5 ("Location") set to NULL. The default value for "Location" is not used since the corresponding array element is defined with a null value. The array elements "7" and "next" do not correspond to column numbers in the "Employee" table, therefore the query ignores them. Here's the **UPDATE** statement that uses this array:

ObjectScript

```
&sql(UPDATE Employee
      VALUES :emp('profile',)
      WHERE Employee = 379)
```

Given the above definitions and array values, this statement will update the values of the "Name", "Address", and "Location" fields of the "Employee" row for which Row ID = 379.

However, omitting the subscript entirely results in an SQLCODE -54 error: Array designator (last subscript omitted) expected after **VALUES**.

You may also use an array reference with an UPDATE query that targets multiple rows, for example:

ObjectScript

```
&sql(UPDATE Employee
      VALUES :emp('profile',)
      WHERE Type = 'PART-TIME')
```

A **VALUES** clause variable cannot use dot syntax. Therefore, the following embedded SQL example is correct:

```
SET sname = state.Name
&sql(INSERT INTO StateTbl VALUES :sname)
```

The following is not correct:

```
&sql(INSERT INTO State VALUES :state.Name)
```

NULL and empty string values are different. For further details, see NULL. For backward compatibility, all empty string (") values in older existing data are considered as NULL values. In new data, empty strings are stored in the data as $CHAR(0). Through SQL, NULL is referenced as 'NULL'. For example:

SQL

```
INSERT INTO Sample.Person
(SSN,Name,Home_City) VALUES ('123-45-6789','Doe,John',NULL)
```

Through SQL, empty string is referenced as '' (two single quotes). For example:

SQL

```
INSERT INTO Sample.Person
(SSN,Name,Home_City) VALUES ('123-45-6789','Doe,John','')
```

You cannot insert a NULL value for the ID field.

Arguments

field

A field name or a comma-separated list of field names.

value

A value or comma-separated list of values. Each value is assigned to the corresponding field.

Examples

The following example inserts a record for "Doe,John" into the Sample.Person table. It then selects this record, and then deletes this record. A second **SELECT** confirms the deletion.

SQL

```
INSERT INTO Sample.Person (Name,SSN,Home_City) VALUES ("Doe,John","123-45-6789","Metropolis")
SELECT Name,SSN,Home_City FROM Sample.Person WHERE Name ="Doe,John"
DELETE FROM Sample.Person WHERE Name="Doe,John"
SELECT Name,SSN FROM Sample.Person WHERE Name='Doe,John'
```

See Also

- INSERT

- INSERT OR UPDATE

- UPDATE

- SQLCODE error messages

WHERE (SQL)

A SELECT clause that specifies one or more restrictive conditions.

Synopsis

```
SELECT fields
    FROM table
    WHERE condition-expression
```

Arguments

Argument	Description
condition-expression	An expression consisting of one or more boolean predicates governing which data values are to be retrieved.

Description

The optional **WHERE** clause can be used for the following purposes:

- To specify predicates that restrict which data values are to be returned.

- To specify an explicit join between two tables.

- To specify an implicit join between the base table and a field in another table.

The **WHERE** clause is most commonly used to specify one or more predicates that are used to restrict the data (filter out rows) retrieved by a SELECT query or subquery. You can also use a **WHERE** clause in an UPDATE command, DELETE command, or in a result set **SELECT** in an INSERT (or INSERT OR UPDATE) command.

The **WHERE** clause qualifies or disqualifies specific rows from the query selection. The rows that qualify are those for which the *condition-expression* is true. The *condition-expression* can be one or more logical tests (predicates). Multiple predicates can be linked by the AND and OR logical operators. See "Predicates and Logical Operators" for further details and restrictions.

If a predicate includes division and there are any values in the database that could produce a divisor with a value of zero or a NULL value, you cannot rely on order of evaluation to avoid division by zero. Instead, use a **CASE** statement to suppress the risk.

A **WHERE** clause can specify a *condition-expression* that includes a subquery. The subquery must be enclosed in parentheses.

A **WHERE** clause can specify an explicit join between two tables using the = (inner join) symbolic join operator. For further details, refer to JOIN.

A **WHERE** clause can specify an implicit join between the base table and a field from another table using the arrow syntax (–>) operator. For further details, refer to Implicit Joins.

Specifying a Field

The simplest form of a **WHERE** clause specifies a predicate comparing a field to a value, such as `WHERE Age > 21`. Valid field values include the following: a column name (`WHERE Age > 21`); an %ID, %TABLENAME, or %CLASSNAME; a scalar function specifying a column name (`WHERE ROUND(Age,-1)=60`), a collation function specifying a column name (`WHERE %SQLUPPER(Name) %STARTSWITH ' AB'`).

You cannot specify a field by column number.

Because the name of the RowID field can change when a table is re-compiled, a **WHERE** clause should avoid referring to the RowID by name (for example, WHERE ID=22). Instead, refer to the RowID using the %ID pseudo-column name (for example, WHERE %ID=22).

You cannot specify a field by column alias; attempting to do so generates an SQLCODE -29 error. However, you can use a subquery to define a column alias, then use this alias in the **WHERE** clause. For example:

SQL

```
SELECT Interns FROM
    (SELECT Name AS Interns FROM Sample.Employee WHERE Age<21)
WHERE Interns %STARTSWITH 'A'
```

You cannot specify an aggregate field; attempting to do so generates an SQLCODE -19 error. However, you can supply an aggregate function value to a **WHERE** clause by using a subquery. For example:

SQL

```
SELECT Name,Age,AvgAge
FROM (SELECT Name,Age,AVG(Age) AS AvgAge FROM Sample.Person)
WHERE Age < AvgAge
ORDER BY Age
```

Integers and Strings

If a field defined as integer data type is compared to a numeric value, the numeric value is converted to canonical form before performing the comparison. For example, WHERE Age=007.00 parses as WHERE Age=7. This conversion occurs in all modes.

If a field defined as integer data type is compared to a string value in Display mode, the string is parsed as a numeric value. For instance, an empty string (''), like any non-numeric string, is parsed as the number 0. This parsing follows ObjectScript rules for handling strings as numbers. For example, WHERE Age='twenty' parses as WHERE Age=0; WHERE Age='20something' parses as WHERE Age=20. For further details, refer to Strings as Numbers. SQL only performs this parsing in Display mode; in Logical or ODBC mode comparing an integer to a string value returns null.

To compare a string field with a string containing a single quote, double the single quote. For example, WHERE Name %STARTSWITH 'O''' returns O'Neil and O'Connor, but not Obama.

Date and Time

In InterSystems SQL, dates and times are compared and stored using a Logical Mode internal representation. They can be returned in Logical mode, Display Mode, or ODBC mode. For example, the date September 28, 1944 is represented as: Logical mode 37891, Display mode 09/28/1944, ODBC mode 1944-09-28. When specifying a date or time in a *condition-expression*, an error can occur due to a mismatch of SQL mode and date or time format, or due to an invalid date or time value.

A **WHERE** clause *condition-expression* must use the date or time format that corresponds to the current mode. For example, when in Logical mode, to return records with a date of birth in 2005, the **WHERE** clause would appear as follows: WHERE DOB BETWEEN 59901 AND 60265. When in Display mode, the same **WHERE** clause would appear as follows: WHERE DOB BETWEEN '01/01/2005' AND '12/31/2005'.

Failing to match the *condition-expression* date or time format to the display mode results in an error:

- In Display mode or ODBC mode, specifying date data in the incorrect format generates an SQLCODE -146 error. Specifying time data in the incorrect format generates an SQLCODE -147 error.

- In Logical mode, specifying date or time data in the incorrect format does not generate an error, but either returns no data or returns unintended data. This is because Logical mode does not parse a date or time in Display or ODBC format as a date or time value. The following **WHERE** clause, when executed in Logical mode, returns unintended data: WHERE DOB BETWEEN 37500 AND 38000 AND DOB <> '1944-09-28' returns a range of DOB values, including DOB=37891 (September 28, 1944), which the <> predicate was attempting to omit.

An invalid date or time value also generates an SQLCODE -146 or -147 error. An invalid date is one that you can specify in Display mode/ODBC mode, but InterSystems IRIS cannot convert into a Logical mode equivalent. For example, in ODBC mode the following generates an SQLCODE -146 error: `WHERE DOB > '1830-01-01'` because InterSystems IRIS cannot process a date value prior to December 31, 1840. The following in ODBC mode also generates an SQLCODE -146 error: `WHERE DOB BETWEEN '2005-01-01' AND '2005-02-29'`, because 2005 is not a leap year.

When in Logical mode, a Display mode or ODBC mode value is not parsed as a date or time value, and therefore its value is not validated. For this reason, in Logical mode a **WHERE** clause such as `WHERE DOB > '1830-01-01'` does not return an error.

Stream Fields

In most situations, you cannot use a stream field in a **WHERE** clause predicate. Doing so results in an SQLCODE -313 error. However, the following uses of stream fields are allowed in a **WHERE** clause:

- Stream null testing: you can specify the predicate `streamfield IS NULL` or `streamfield IS NOT NULL`.

- Stream length testing: you can specify a `CHARACTER_LENGTH(streamfield)`, `CHAR_LENGTH(streamfield)`, or `DATALENGTH(streamfield)` function in a **WHERE** clause predicate.

- Stream substring testing: you can specify a `SUBSTRING(streamfield,start,length)` function in a **WHERE** clause predicate.

List Structures

InterSystems IRIS supports the list structure data type %List (data type class %Library.List). This is a compressed binary format, which does not map to a corresponding native data type for InterSystems SQL. It corresponds to data type VARBINARY with a default MAXLEN of 32749. For this reason, Dynamic SQL cannot use %List data in a **WHERE** clause comparison. For further details, refer to Data Types.

To reference structured list data, use the **%INLIST** predicate or the **FOR SOME %ELEMENT** predicate.

To use the data values of a list field in a *condition-expression*, you can use **%EXTERNAL** to compare the list values to a predicate. For example, to return all records in which the FavoriteColors list field value consists of the single element 'Red':

SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE %EXTERNAL(FavoriteColors)='Red'
```

When **%EXTERNAL** converts a list to DISPLAY format, the displayed list items appear to be separated by a blank space. This "space" is actually the two non-display characters CHAR(13) and CHAR(10). To use a *condition-expression* against more than one element in the list, you must specify these characters. For example, to return all records in which the FavoriteColors list field value consists of the two elements 'Orange' and 'Black' (in that order):

SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE %EXTERNAL(FavoriteColors)='Orange'||CHAR(13)||CHAR(10)||'Black'
```

Variables

A **WHERE** clause predicate can specify:

A %TABLENAME, or %CLASSNAME pseudo-field variable keyword. %TABLENAME returns the current table name. %CLASSNAME returns the name of the class corresponding to the current table. If the query references multiple tables, you can prefix the keyword with a table alias. For example, `t1.%TABLENAME`.

One or more of the following ObjectScript special variables (or their abbreviations): $HOROLOG, $JOB, $NAMESPACE, $TLEVEL, $USERNAME, $ZHOROLOG, $ZJOB, $ZNSPACE, $ZPI, $ZTIMESTAMP, $ZTIMEZONE, $ZVERSION.

List of Predicates

The SQL predicates fall into the following categories:

* Equality Comparison Predicates

* BETWEEN Predicate

* IN and %INLIST Predicates

* %STARTSWITH Predicate and Contains Operator

* NULL Predicate

* EXISTS Predicate

* FOR SOME Predicate

* FOR SOME %ELEMENT Predicate

* LIKE, %MATCHES, and %PATTERN Predicates

* %INSET and %FIND Predicates

Predicate Case-Sensitivity

A predicate uses the collation type defined for the field. By default, string data type fields are defined with SQLUPPER collation, which is not case-sensitive. You can define the string collation default for the current namespace and specify a non-default field collation type when defining a field/property.

The **%INLIST**, Contains operator ([), **%MATCHES**, and **%PATTERN** predicates do not use the field's default collation. They always uses EXACT collation, which is case-sensitive.

A predicate comparison of two literal strings is always case-sensitive.

Predicate Conditions and %NOINDEX

You can preface a predicate condition with the %NOINDEX keyword to prevent the query optimizer using an index on that condition. This is most useful when specifying a range condition that is satisfied by the vast majority of the rows. For example, `WHERE %NOINDEX Age >= 1`. For further details, refer to Using %ALLINDEX, %IGNOREINDEX, and %NOINDEX.

Predicate Condition on Outlier Value

If the **WHERE** clause in a Dynamic SQL query selects on a non-null outlier value, you can significantly improve performance by enclosing the outlier value literal in double parentheses. These double parentheses cause Dynamic SQL to use the outlier selectivity when optimizing. For example, if your business is located in Massachusetts (MA), a large percentage of your employees will reside in Massachusetts. For the Employees table Home_State field, `'MA'` is the outlier value. To optimally select for this value, you should specify `WHERE Home_State=(('MA'))`.

This syntax should not be used in Embedded SQL or in a view definition. In Embedded SQL or a view definition, the outlier selectivity is always used and requires no special coding.

A **WHERE** clause in a Dynamic SQL query automatically optimizes for a null outlier value. For example, a clause such as `WHERE FavoriteColors IS NULL`. No special coding is required for IS NULL and IS NOT NULL predicates when NULL is the outlier value.

Outlier selectivity is determined by running the Tune Table utility.

Equality Comparison Predicates

The following are the available equality comparison predicates:

Table C–2: SQL Equality Comparison Predicates

Predicate	Operation
=	Equals
<>	Does not equal
!=	Does not equal
>	Is greater than
<	Is less than
>=	Is greater than or equal to
<=	Is less than or equal to

For example:

SQL

```
SELECT Name, Age FROM Sample.Person
WHERE Age < 21
```

SQL defines comparison operations in terms of collation: the order in which values are sorted. Two values are equal if they collate in exactly the same way. A value is greater than another value if it collates after the second value. String field collation takes the field's default collation. The InterSystems IRIS default collation is not case-sensitive. Thus, a comparison of two string field values or a comparison of a string field value with a string literal is (by default) not case-sensitive. For example, if Home_State field values are uppercase two-letter strings:

Expression	Value
'MA' = Home_State	TRUE for values MA.
'ma' = Home_State	TRUE for values MA.
'VA' < Home_State	TRUE for values VT, WA, WI, WV, WY.
'ar' >= Home_State	TRUE for values AK, AL, AR.

Note, however, that a comparison of two literal strings *is* case-sensitive: `WHERE 'ma' = 'MA'` is always FALSE.

BETWEEN Predicate

The **BETWEEN** comparison operator allows you to select those data values that are in the range specified by the syntax BETWEEN *lowval* AND *highval*. This range is inclusive of the *lowval* and *highval* values themselves. This is equivalent to a paired greater than or equal to operator and a less than or equal to operator. This comparison is shown in the following example:

SQL

```
SELECT Name,Age FROM Sample.Person
WHERE Age BETWEEN 18 AND 21
```

This returns all the records in the Sample.Person table with an Age value between 18 and 21, inclusive of those values. Note that you must specify the BETWEEN values in ascending order; a predicate such as `BETWEEN 21 AND 18` would return no records.

Like most predicates, BETWEEN can be inverted using the NOT logical operator, as shown in the following example:

SQL

```
SELECT Name,Age FROM Sample.Person
WHERE Age NOT BETWEEN 20 AND 55
ORDER BY Age
```

This returns all the records in the Sample.Person table with an Age value less than 20 or greater than 55, exclusive of those values.

BETWEEN is commonly used for a range of numeric values, which collate in numeric order. However, **BETWEEN** can be used for a collation sequence range of values of any data type.

BETWEEN uses the same collation type as the column it is matching against. By default, string data types collate as not case-sensitive.

For further details, refer to the BETWEEN predicate.

IN and %INLIST Predicates

The IN predicate is used for matching a value to an unstructured series of items. It has the following syntax:

```
WHERE field IN (item1,item2[,...])
```

Collation applies to the IN comparison as it applies to an equality test. IN uses the field's default collation. By default, comparisons with field string values are not case-sensitive.

The %INLIST predicate is an InterSystems IRIS extension for matching a value to the elements of an InterSystems IRIS list structure. It has the following syntax:

```
WHERE item %INLIST listfield
```

%INLIST uses EXACT collation. Therefore, by default, **%INLIST** string comparisons are case-sensitive.

With either predicate you can perform equality comparisons and subquery comparisons.

For further details, refer to IN and %INLIST.

Substring Predicates

You can use the following to compare a field value to a substring:

Table C–3: SQL Substring Predicates

Predicate	Operation
%STARTSWITH	The value must start with the specified substring.
[	Contains operator. The value must contain the specified substring.

%STARTSWITH Predicate

The InterSystems IRIS **%STARTSWITH** comparison operator permits you to perform partial matching on the initial characters of a string or numeric. The following example uses **%STARTSWITH**. to select those records in which the Name value begins with "S":

SQL

```
SELECT Name,Age FROM Sample.Person
WHERE Name %STARTSWITH 'S'
```

Like other string field comparisons, **%STARTSWITH** comparisons use the field's default collation. By default, string fields are not case-sensitive. For example:

SQL

```
SELECT Name,Home_City,Home_State FROM Sample.Person
WHERE Home_City %STARTSWITH Home_State
```

For further details, refer to %STARTSWITH.

Contains Operator ([)

The Contains operator is the open bracket symbol: [. It permits you to match a substring (string or numeric) to any part of a field value. The comparison is always case-sensitive. The following example uses the Contains operator to select those records in which the Name value contains a "S":

SQL

```
SELECT Name, Age FROM Sample.Person
WHERE Name [ 'S'
```

NULL Predicate

This detects undefined values. You can detect all null values, or all non-null values. The NULL predicate has the following syntax:

```
WHERE field IS [NOT] NULL
```

NULL predicate conditions are one of the few predicates that can be used on stream fields in a WHERE clause.

For further details, refer to NULL.

EXISTS Predicate

This operates with subqueries to test whether a subquery evaluates to the empty set.

SQL

```
SELECT t1.disease FROM illness_tab t1 WHERE EXISTS
 (SELECT t2.disease FROM disease_registry t2
 WHERE t1.disease = t2.disease
 HAVING COUNT(t2.disease) > 100)
```

For further details, refer to EXISTS.

FOR SOME Predicate

The FOR SOME predicate of the WHERE clause can be used to determine whether or not to return any records based on a condition test of one or more field values. This predicate has the following syntax:

```
FOR SOME (table [AS t-alias]) (fieldcondition)
```

FOR SOME specifies that *fieldcondition* must evaluate to true; at least one of the field values must match the specified condition. *table* can be a single table or a comma-separated list of tables, and each table can optionally take a table alias. *fieldcondition* specifies one or more conditions for one or more fields within the specified *table*. Both the *table* argument and the *fieldcondition* argument must be delimited by parentheses.

The following example shows the use of the FOR SOME predicate to determine whether to return a result set:

SQL

```
SELECT Name,Age AS AgeWithWorkers
FROM Sample.Person
WHERE FOR SOME (Sample.Person) (Age<65)
ORDER BY Age
```

In the above example, if at least one field contains an Age value less than the specified age, all of the records are returned. Otherwise, no records are returned.

For further details, refer to FOR SOME.

FOR SOME %ELEMENT Predicate

The **FOR SOME %ELEMENT** predicate of the WHERE clause has the following syntax:

```
FOR SOME %ELEMENT(field) [AS e-alias] (predicate)
```

The **FOR SOME %ELEMENT** predicate matches the elements in *field* with the specified *predicate* clause value. The SOME keyword specifies that at least one of the elements in *field* must satisfy the specified *predicate* condition. The *predicate* can contain the %VALUE or %KEY keyword.

The **FOR SOME %ELEMENT** predicate is a Collection Predicate.

For further details, refer to FOR SOME %ELEMENT.

LIKE, %MATCHES, and %PATTERN Predicates

These three predicates allow you to perform pattern matching.

- LIKE allows you to pattern match using literals and wildcards. Use LIKE when you wish to return data values that contain a known substring of literal characters, or contain several known substrings in a known sequence. LIKE uses the collation of its target for letter case comparisons.

- %MATCHES allows you to pattern match using literals, wildcards, and lists and ranges. Use %MATCHES when you wish to return data values that contain a known substring of literal characters, or contain one or more literal characters that fall within a list or range of possible characters, or contain several such substrings in a known sequence. %MATCHES uses EXACT collation for letter case comparisons.

- %PATTERN allows you to specify a pattern of character types. For example, '1U4L1",".A' (1 uppercase letter, 4 lowercase letters, one literal comma, followed by any number of letter characters of either case). Use %PATTERN when you wish to return data values that contain a known sequence of character types. %PATTERN can specify known literal characters, but is especially useful when the data value is unimportant, but the character type format of those values is significant.

To perform a comparison with the first characters of a string, use the %STARTSWITH predicate.

Predicates and Logical Operators

Multiple predicates can be associated using the AND and OR logical operators. Multiple predicates can be grouped using parentheses. Because InterSystems IRIS optimizes execution of the **WHERE** clause using defined indexes and other optimizations, the order of evaluation of predicates linked by AND and OR logical operators cannot be predicted. For this reason, the order in which you specify multiple predicates has little or no effect on performance. If strict left-to-right evaluation of predicates is desired, you can use a CASE statement.

Note: The OR logical operator cannot be used to associate a FOR SOME %ELEMENT collection predicate that references a table field with a predicate that a references a field in a different table. For example,

```
WHERE FOR SOME %ELEMENT(t1.FavoriteColors) (%VALUE='purple')
OR t2.Age < 65
```

Because this restriction depends on how the optimizer uses indexes, SQL may only enforce this restriction when indexes are added to a table. It is strongly suggested that this type of logic be avoided in all queries.

For further details, refer to Logical Operators.

See Also

- **SELECT** statement
- **HAVING** clause
- **Overview of Predicates**
- **Querying the Database**
- **SQLCODE error messages**

WHERE CURRENT OF (SQL)

An UPDATE/DELETE clause that specifies the current row using a cursor.

Synopsis

```
WHERE CURRENT OF cursor
```

Description

The **WHERE CURRENT OF** clause can be used in a cursor-based Embedded SQL **UPDATE** or **DELETE** statement to specify the cursor positioned on the record to be updated or deleted. For example:

ObjectScript

```
&sql(DELETE FROM Sample.Employees WHERE CURRENT OF EmployeeCursor)
```

which deletes the row that the last **FETCH** command obtained from the "EmployeeCursor" cursor.

An Embedded SQL **UPDATE** or **DELETE** can use a WHERE clause (with no cursor), or a **WHERE CURRENT OF** with a declared cursor, but not both. If you specify an **UPDATE** or **DELETE** with neither **WHERE** nor **WHERE CURRENT OF**, all of the records in the table are updated or deleted.

UPDATE Restriction

When using a **WHERE CURRENT OF** clause, you cannot update a field using the current field value to generate an updated value. For example, SET Salary=Salary+100 or SET Name=UPPER(Name). Attempting to do so results in an SQLCODE -69 error: SET <field> = <value expression> not allowed with WHERE CURRENT OF <cursor>.

Arguments

cursor

Specifies that the operation is done at the current position of *cursor*, which is a cursor that points to the table.

Examples

The following Embedded SQL example shows an **UPDATE** operation using **WHERE CURRENT OF**:

ObjectScript

```
 NEW %ROWCOUNT,%ROWID
 &sql(DECLARE WPCursor CURSOR FOR
      SELECT Lang FROM SQLUser.WordPairs
      WHERE Lang='Sp')
 &sql(OPEN WPCursor)
     QUIT:(SQLCODE'=0)
  FOR { &sql(FETCH WPCursor)
     QUIT:SQLCODE
     &sql(UPDATE SQLUser.WordPairs SET Lang='Es'
    WHERE CURRENT OF WPCursor)
  IF SQLCODE=0 {
  WRITE !,"Update succeeded"
  WRITE !,"Row count=",%ROWCOUNT," RowID=",%ROWID }
  ELSE {
  WRITE !,"Update failed, SQLCODE=",SQLCODE }
  }
  &sql(CLOSE WPCursor)
```

The following Embedded SQL example shows a **DELETE** operation using **WHERE CURRENT OF**:

ObjectScript

```
NEW %ROWCOUNT,%ROWID
 &sql(DECLARE WPCursor CURSOR FOR
       SELECT Lang FROM SQLUser.WordPairs
       WHERE Lang='En')
 &sql(OPEN WPCursor)
       QUIT:(SQLCODE'=0)
 FOR { &sql(FETCH WPCursor)
       QUIT:SQLCODE
       &sql(DELETE FROM SQLUser.WordPairs
       WHERE CURRENT OF WPCursor)
   IF SQLCODE=0 {
   WRITE !,"Delete succeeded"
   WRITE !,"Row count=",%ROWCOUNT," RowID=",%ROWID }
   ELSE {
   WRITE !,"Delete failed, SQLCODE=",SQLCODE }
   }
   &sql(CLOSE WPCursor)
```

See Also

- DECLARE, OPEN, FETCH, CLOSE

- DELETE, UPDATE, INSERT OR UPDATE

- SQL Cursors

- SQLCODE error messages

SQL Predicate Conditions

Overview of Predicates

Describes logical conditions that evaluate to either true or false.

Use of Predicates

A predicate is a condition expression that evaluates to a boolean value, either true or false.

Predicates can be used as follows:

- In a SELECT statement's WHERE clause or HAVING clause to determine which rows are relevant to a particular query. Note that not all predicates can be used in a **HAVING** clause.

- In a JOIN operation's **ON** clause to determine which rows are relevant to the join operation.

- In an UPDATE or DELETE statement's **WHERE** clause, to determine which rows are to be modified.

- In a WHERE CURRENT OF statement's **AND** clause.

- In a CREATE TRIGGER statement's **WHEN** clause to determine when to apply triggered action code.

- In a **DROP** statement, such as **DROP TABLE**, to suppress errors occurring if the target does not exist.

List of Predicates

Every predicate contains one or more comparison operators, either symbols or keyword clauses. InterSystems SQL supports the following comparison operators:

Comparison Operator	Description
= (equals) <> (does not equal) != (does not equal) > (is greater than) >= (is greater than or equal to) < (is less than) <= (is less than or equal to)	Equality comparison conditions. Can be used for numeric comparisons or string collation order comparisons. For numeric comparisons, an empty string value ('') is evaluated as 0. A NULL in any equality comparison always returns the empty set; use the **IS NULL** predicate instead. See Relational Operators.
IS [NOT] NULL	Tests whether a field has undefined (NULL) values. See IS NULL.
IS [NOT] JSON	Tests whether a value is a JSON formatted string or an OREF to a JSON array or a JSON object. See IS JSON.
EXISTS (subquery)	Uses a subquery to test a specified table for existence of one or more rows. See EXISTS.
DROP-command IF EXISTS *objectname*	Conditions the execution of a **DROP** command on the existence of the specified target, suppressing the error if it does not exist. See EXISTS.
BETWEEN x AND y	A **BETWEEN** condition uses >= and <= comparison conditions together. Match must be between two specified range limit values (inclusive). See BETWEEN.

Comparison Operator	Description
IN (item1,item2[...,itemn]) IN (subquery)	An equality condition that matches a field value to any of the items in a comma-separated list, or any of the items returned by a subquery. See IN.
%INLIST listfield	An equality condition that matches a field value to any of the elements in a %List structured list. See %INLIST.
[	Contains operator. Match must contain the specified string. The Contains operator uses EXACT collation, and is therefore case-sensitive. Must specify value in Logical format.
]	Follows operator. Match must appear after the specified item in collation sequence. Must specify value in Logical format.
%STARTSWITH string	Match must begin with the specified string. See %STARTSWITH.
FOR SOME	A boolean comparison condition. The **FOR SOME** condition must be true for at least one data value of the specified field. See FOR SOME.
FOR SOME %ELEMENT	A list element comparison condition with a %VALUE or %KEY predicate clause. %VALUE must match the value of at least one element of the list. %KEY must be less than or equal to the number of elements in the list. %VALUE and %KEY clauses can use any of the other comparison operators. See FOR SOME %ELEMENT.
LIKE	A pattern match condition using literals and wildcards. Use **LIKE** when you wish to return data values that contain a known substring of literal characters, or contain several known substrings in a known sequence. **LIKE** uses the collation of its target for letter case comparisons. (Contrast with the Contains operator, which uses EXACT collation.) See LIKE.
%MATCHES	A pattern match condition using literals, wildcards, and lists and ranges. Use **%MATCHES** when you wish to return data values that contain a known substring of literal characters, or contain one or more literal characters that fall within a list or range of possible characters, or contain several such substrings in a known sequence. **%MATCHES** uses EXACT collation for letter case comparisons. See %MATCHES.

Comparison Operator	Description
%PATTERN	A pattern match condition using character types. For example, `'1U4L1",".A'` (1 uppercase letter, 4 lowercase letters, one literal comma, followed by any number of letter characters of either case). Use **%PATTERN** when you wish to return data values that contain a known sequence of character types. **%PATTERN** can specify known literal characters, but is especially useful when the data value is unimportant, but the character type format of those values is significant. See %PATTERN.
ALL ANY SOME	A quantified-comparison condition. See ALL, ANY, and SOME.
%INSET %FIND	Field value comparison conditions that enable filtering of RowId field values using an abstract, programmatically specified temp-file or bitmap index. %INSET supports simple comparisons. %FIND supports comparisons involving a bitmap index.

NULL

A NULL is the absence of any value. By definition, it fails all boolean tests: no value is equal to NULL, no value is unequal to NULL, no value is greater than or less than NULL. Even NULL=NULL fails as a predicate. Because the IN predicate is a series of OR'ed equality tests, it is not meaningful to specify NULL in the IN value list. Therefore, specifying any predicate condition eliminates any instances of that field that are NULL. The only way to include NULL fields in the result set from a predicate condition is to use the IS NULL predicate. This is shown in the following example:

SQL

```
SELECT FavoriteColors FROM Sample.Person
WHERE FavoriteColors = $LISTBUILD('Red') OR FavoriteColors IS NULL
```

Collation

A predicate uses the collation type defined for the field. By default, string data type fields are defined with SQLUPPER collation, which is not case-sensitive. You can define the string collation default for the current namespace and specify a non-default field collation type when defining a field/property.

If you specify a collation type in a query, you must specify it on both sides of the comparison. Specifying a collation type can affect index usage; for further details, refer to Index Collation.

Certain predicate comparisons can involve substrings embedded within a string: the Contains operator ([), the **%MATCHES** predicate, and the **%PATTERN** predicate. These predicates always uses EXACT collation, and are therefore always case-sensitive. Because some collations prepend a blank space to a string, these predicates could not perform their function if they followed the field's default collation. However, the **LIKE** predicate can use wildcards to match substrings embedded within a string. **LIKE** uses the field's default collation, which by default is not case-sensitive.

Compound Predicates

A predicate is the simplest version of a condition expression; a condition expression can consist of one or more predicates. You can link multiple predicates together with the AND and OR logical operators. You can invert the sense of a predicate

by placing the NOT unary operator before the predicate. The NOT unary operator only affects the predicate that immediately follows it. Predicates are evaluated in strict left-to-right order. You can use parentheses to group predicates. You can place a NOT unary operator before the opening parentheses to invert the sense of a group of predicates. Spaces are not required before or after parentheses, or between parentheses and logical operators.

The **IN** and **%INLIST** predicates are functionally equivalent to multiple OR equality predicates. The following examples are equivalent:

ObjectScript

```
SET q1="SELECT Name,Home_State FROM Sample.Person "
SET q2="WHERE Home_State='MA' OR Home_State='VT' OR Home_State='NH'"
SET myquery=q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
```

ObjectScript

```
SET q1="SELECT Name,Home_State FROM Sample.Person "
SET q2="WHERE Home_State IN('MA','VT','NH')"
SET myquery=q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
```

ObjectScript

```
SET list=$LISTBUILD("MA","VT","NH")
SET q1="SELECT Name,Home_State FROM Sample.Person "
SET q2="WHERE Home_State %INLIST(?)"
SET myquery=q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(list)
DO rset.%Display()
```

The **FOR SOME %ELEMENT** predicate can contain logical operators, as well as be linked to other predicates using logical operators. This is shown in the following example:

SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FOR SOME %ELEMENT(FavoriteColors)(%VALUE='Red' OR %Value='White'
      OR %Value %STARTSWITH 'B')
      AND (Name BETWEEN 'A' AND 'F' OR Name %STARTSWITH 'S')
ORDER BY Name
```

Note the parentheses around `(Name BETWEEN 'A' AND 'F' OR Name %STARTSWITH 'S')`; without these grouping parentheses, the **FOR SOME %ELEMENT** condition would not apply to `Name %STARTSWITH 'S'`.

Collection Predicates with OR

FOR SOME %ELEMENT is a Collection Predicate. The use of this predicate with the OR logical operator is restricted, as follows. The OR logical operator cannot be used to associate a Collection Predicate that references a table field with a predicate that a references a field in a different table. For example,

```
WHERE FOR SOME %ELEMENT(t1.FavoriteColors) (%VALUE='purple')
OR t2.Age < 65
```

Because this restriction depends on how the optimizer uses indexes, SQL may only enforce this restriction when indexes are added to a table. It is strongly suggested that this type of logic be avoided in all queries.

Predicates and %SelectMode

All predicates perform their comparisons using Logical (internal storage) data values. However, some predicates can perform format mode conversion on the predicate value(s), converting it from ODBC or Display format to Logical format. Other predicates cannot perform format mode conversion, and therefore must always specify the predicate value in Logical format.

Predicates that perform format mode conversion determine whether conversion is required from the data type (such as DATE or %List) of the matching field and determine the type of conversion from the %SelectMode setting. If %SelectMode is set to a value other than Logical format (such as %SelectMode=ODBC or %SelectMode=Display) the predicate value(s) must be specified in the correct ODBC or Display format.

- Equality predicates perform format mode conversion. InterSystems IRIS converts the predicate value to Logical format, then matches it with the field values. If %SelectMode is set to a mode other than Logical format, the predicate value(s) must be specified in the %SelectMode format (ODBC or Display) for data types whose display value differs from the Logical storage value. For example, dates, times, and %List-formatted strings. Because InterSystems IRIS automatically performs this format conversion, specifying this type of predicate value in Logical format commonly results in an SQLCODE error. For example, SQLCODE -146 "Unable to convert date input to a valid logical date value" (InterSystems IRIS assumes the supplied Logical value is an ODBC or Display value and attempts to convert it to a Logical value — which doesn't succeed.) Affected predicates include =, <, >, BETWEEN, and IN.

- Pattern predicates cannot perform format mode conversion, because InterSystems IRIS cannot meaningfully convert the predicate value. Therefore, the predicate value must be specified in Logical format, regardless of the %SelectMode setting. Specifying predicate value(s) in ODBC or Display format commonly results in no data matches or unintended data matches. Affected predicates include %INLIST, LIKE, %MATCHES, %PATTERN, %STARTSWITH, [(the Contains operator), and] (the Follows operator).

You can use the %INTERNAL, %EXTERNAL, or %ODBCOUT format-transform functions to transform the field that the predicate operates upon. This allows you to specify the predicate value in another format. For example, `WHERE %ODBCOut(DOB) %STARTSWITH '1955-'`. However, specifying a format-transform function on a matching field prevents the use of an index for the field. This can have a significant negative effect upon performance.

In the following Dynamic SQL example, the **BETWEEN** predicate (an equality predicate) must specify dates in %SelectMode=1 (ODBC) format:

ObjectScript

```
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE DOB BETWEEN '1950-01-01' AND '1960-01-01'"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

In the following Dynamic SQL examples, the **%STARTSWITH** predicate (a pattern predicate) cannot perform format mode conversion. The first example attempts to specify a **%STARTSWITH** for dates in the %SelectMode=ODBC format for years in the 1950s. However, because the table does not contain birth dates that begin with $HOROLOG 195 (dates in the year 1894), no rows are selected:

ObjectScript

```
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE DOB %STARTSWITH '195'"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

The following example uses the **%ODBCOut** format-transform function on the matching DOB field so that **%STARTSWITH** can be used to select for years in the 1950s in ODBC format. However, note that this usage prevents the use of an index on the DOB field.

ObjectScript

```
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE %ODBCOut(DOB) %STARTSWITH '195'"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

In the following example the **%STARTSWITH** predicate specifies a %STARTSWITH for dates in Logical (internal) format. Rows with DOB Logical values beginning with 41 (dates from April 4 1953 ($HOROLOG 41000) through December 28 1955 ($HOROLOG 41999)) are selected. The DOB field index is used:

ObjectScript

```
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE DOB %STARTSWITH '41'"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

Predicates and PosixTime, Timestamp, and Date

Equality predicate comparisons automatically perform conversion between these different date and datetime representations. This conversion is independent of %SelectMode. Therefore, the following are all meaningful comparison predicates:

```
WHERE MyPosixField = MyTimestampField
WHERE MyPosixField < CURRENT_TIMESTAMP
WHERE MyPosixField BETWEEN DATEADD('month',-1,CURRENT_TIMESTAMP) AND $HOROLOG
WHERE MyPosixField BETWEEN DATEADD('day',-1,CURRENT_DATE) AND LAST_DAY(CURRENT_DATE)
```

Pattern predicate comparisons, such as **%STARTSWITH**, do not perform conversion between different date and datetime representations. The operate on the actual stored data value.

Suppress Literal Substitution

You can literal substitution during compile pre-parsing by enclosing the predicate argument in double parentheses. For example, LIKE(('abc%')). This may improve query performance by improving overall selectivity and/or subscript bounding selectivity. However, it should be avoided when the same query is called multiple times with different values, as it will result in the creation of a separate cached query for each query call.

Example

The following example uses a variety of conditions in the **WHERE** clause of a query:

SQL

```
SELECT PurchaseOrder FROM MyTable
   WHERE OrderTotal >= 1000
   AND ItemName %STARTSWITH :partname
   AND AnnualOrders BETWEEN 50000 AND 100000
   AND City LIKE 'Ch%'
   AND CustomerNumber IN
      (SELECT CustNum FROM TheTop100
       WHERE TheTop100.City='Boston')
   AND :minorder > SOME
      (SELECT OrderTotal FROM Orders
       WHERE Orders.Customer = :cust)
```

See Also

* SELECT statement, HAVING clause, WHERE clause

* CREATE TRIGGER

ALL (SQL)

Matches a value with all corresponding values from a subquery.

Synopsis

```
scalar-expression comparison-operator ALL (subquery)
```

Description

The **ALL** keyword works in conjunction with a comparison operator to create a predicate (a quantified comparison condition) that is true if the value of a scalar expression matches *all* of the corresponding values retrieved by the subquery. The **ALL** predicate compares a single *scalar-expression* item with a single subquery **SELECT** item. A subquery with more than one select item generates an SQLCODE -10 error.

ALL can be used wherever a predicate condition can be specified, as described in Overview of Predicates.

Where applicable, the system automatically applies Set-Valued Subquery Optimization (SVSO) to an **ALL** subquery. For details on this optimization, and using the %NOSVSO keyword to override it, refer to "Query Optimization Options" on the FROM clause reference page.

Arguments

scalar-expression

A scalar expression (most commonly a data column) whose values are being compared with the result set generated by the *subquery*.

comparison-operator

One of the following comparison operators: = (equal to), <> or != (not equal to), < (less than), <= (less than or equal to), > (greater than), >= (greater than or equal to), [(contains), or] (follows).

subquery

A subquery, enclosed in parentheses, which returns a result set from a single column that is used for the comparison with *scalar-expression*.

Examples

The following example selects those ages in the Person database that are less than all of the ages in the Employee database:

SQL

```
SELECT DISTINCT Age FROM Sample.Person
WHERE Age < ALL
   (SELECT Age FROM Sample.Employee)
ORDER BY Age
```

The following example selects those names in the Person database that are longer or shorter than all of the names in the Employee database:

SQL

```
SELECT $LENGTH(Name) AS NameLength,Name FROM Sample.Person
WHERE $LENGTH(Name) > ALL
     (SELECT $LENGTH(Name) FROM Sample.Employee)
OR $LENGTH(Name) < ALL
     (SELECT $LENGTH(Name) FROM Sample.Employee)
```

The following example returns a list of states west of the Mississippi River, all of which states do not contain an employee with the title of Manager or Director:

SQL

```
SELECT DISTINCT State
FROM Sample.USZipCode
WHERE Longitude < -93
  AND State != ALL
    (SELECT Home_State FROM Sample.Employee
     WHERE Title [ 'Manager' OR Title [ 'Director')
ORDER BY State
```

See Also

- SELECT statement, HAVING clause, WHERE clause

- ANY predicate condition

- SOME predicate condition

- Overview of Predicates

ANY (SQL)

Matches a value with at least one matching value from a subquery.

Synopsis

```
scalar-expression comparison-operator ANY (subquery)
```

Description

The **ANY** keyword works in conjunction with a comparison operator to create a predicate (a quantified comparison condition) that is true if the value of a scalar expression matches one or more of the corresponding values retrieved by the subquery. The **ANY** predicate compares a single *scalar-expression* item with a single subquery **SELECT** item. A subquery with more than one select item generates an SQLCODE -10 error.

Note: The **ANY** and **SOME** keywords are synonyms.

ANY can be used wherever a predicate condition can be specified, as described in Overview of Predicates.

Where applicable, the system automatically applies Set-Valued Subquery Optimization (SVSO) to an **ANY** subquery. For details on this optimization, and using the %NOSVSO keyword to override it, refer to "Query Optimization Options" on the FROM clause reference page.

Arguments

scalar-expression

A scalar expression (most commonly a data column) whose values are being compared with the result set generated by *subquery*.

comparison-operator

One of the following comparison operators: = (equal to), <> or != (not equal to), < (less than), <= (less than or equal to), > (greater than), >= (greater than or equal to), [(contains), or] (follows).

subquery

A subquery, enclosed in parentheses, which returns a result set that is used for the comparison with *scalar-expression*.

Example

The following example selects those employees with salaries greater than $75,000 that live in any of the states west of the Mississippi River:

SQL

```
SELECT Name,Salary,Home_State FROM Sample.Employee
WHERE Salary > 75000
AND Home_State = ANY
 (SELECT State FROM Sample.USZipCode
  WHERE Longitude < -93)
ORDER BY Home_State
```

See Also

* SELECT statement, HAVING clause, WHERE clause

* ALL predicate condition

* SOME predicate condition

- [Overview of Predicates](#)

BETWEEN (SQL)

Matches a value to a range of values.

Synopsis

```
scalar-expression BETWEEN lowval AND highval
```

Description

The **BETWEEN** predicate allows you to select those data values that are in the range specified by *lowval* and *highval*. This range is inclusive of the *lowval* and *highval* values themselves. This is equivalent to a paired greater than or equal to operator and a less than or equal to operator. This comparison is shown in the following example:

SQL

```
SELECT Name,Age FROM Sample.Person
WHERE Age BETWEEN 18 AND 21
ORDER BY Age
```

This returns all the records in the Sample.Person table with an Age value between 18 and 21, inclusive of those values. Note that you must specify the **BETWEEN** values in ascending order; a predicate such as `BETWEEN 21 AND 18` would return the null string. If none of the scalar expression values fall within the specified range, **BETWEEN** returns the null string.

Like most predicates, **BETWEEN** can be inverted using the NOT logical operator. Neither **BETWEEN** nor **NOT BETWEEN** can be used to return NULL fields. To return NULL fields use IS NULL. **NOT BETWEEN** is shown in the following example:

SQL

```
SELECT Name,Age FROM Sample.Person
WHERE Age NOT BETWEEN 20 AND 55
ORDER BY Age
```

This returns all the records in the Sample.Person table with an Age value less than 20 or greater than 55, exclusive of those values.

BETWEEN can be used wherever a predicate condition can be specified, as described in Overview of Predicates.

Collation Types

BETWEEN is commonly used for a range of numeric values, which collate in numeric order. However, **BETWEEN** can be used for a collation sequence range of values of any data type.

BETWEEN uses the same collation type as the column it is matching against. By default, string data types collate as SQLUPPER, which is not case-sensitive. The "Collation" provides details on defining the string collation default for the current namespace and specifying a non-default field collation type when defining a field/property.

If your query assigns a different collation type to the column, you must also apply this collation type to the **BETWEEN** *substring*. This is shown in the following examples:

In the following example, **BETWEEN** uses the fields' default letter case collation, SQLUPPER, which is not case-sensitive. It returns records where Name is higher in alphabetical order than Home_State, and Home_State is higher in alphabetical order than Home_City:

SQL

```
SELECT Name,Home_State,Home_City
FROM Sample.Person
WHERE Home_State BETWEEN Name AND Home_City
ORDER BY Home_State
```

In the following example, **BETWEEN** string comparisons are not case-sensitive, because the Home_State field is defined as SQLUPPER. This means that the *lowval* and *highval* are functionally identical, selecting 'MA' in any lettercase:

SQL

```
SELECT Name,Home_State FROM Sample.Person
WHERE Home_State
    BETWEEN 'MA' AND 'Ma'
ORDER BY Home_State
```

In the following example, the %SQLSTRING collation function causes **BETWEEN** string comparisons to be case-sensitive. It selects those records with Home_State values of 'MA' through 'Ma', which in this data set includes 'MA', 'MD', 'ME', 'MO', 'MS', and 'MT':

SQL

```
SELECT Name,Home_State FROM Sample.Person
WHERE %SQLSTRING(Home_State)
    BETWEEN %SQLSTRING('MA') AND %SQLSTRING('Ma')
ORDER BY Home_State
```

In the following example, the **BETWEEN** string comparison is not case-sensitive and ignores blank spaces and punctuation marks:

SQL

```
SELECT Name FROM Sample.Person
WHERE %STRING(Name) BETWEEN %SQLSTRING('OA') AND %SQLSTRING('OZ')
ORDER BY Name
```

Refer to %SQLUPPER for further information on case transformation functions.

The following example shows **BETWEEN** used in an INNER JOIN operation ON clause. It is performing a string comparison which is not case-sensitive:

SQL

```
SELECT P.Name AS PersonName,E.Name AS EmpName
FROM Sample.Person AS P INNER JOIN Sample.Employee AS E
ON P.Name BETWEEN 'an' AND 'ch' AND P.Name=E.Name
```

%SelectMode

If %SelectMode is set to a value other than Logical format, the **BETWEEN** predicate values must be specified in the %SelectMode format (ODBC or Display). This applies mainly to dates, times, and InterSystems IRIS format lists (%List). Specifying predicate value(s) in Logical format commonly results in an SQLCODE error. For example, SQLCODE -146 "Unable to convert date input to a valid logical date value".

In the following Dynamic SQL example, the **BETWEEN** predicate must specify dates in %SelectMode=1 (ODBC) format:

ObjectScript

```
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE DOB BETWEEN '1950-01-01' AND '1960-01-01'"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

Arguments

scalar-expression

A scalar expression (most commonly a data column) whose values are being compared with the range of values between *lowval* and *highval* (inclusive).

expression

Expression that resolves to the low collation sequence value specifying the beginning of a range of values to match with each value in *scalar-expression*.

expression

Expression that resolves to the high collation sequence value specifying the end of a range of values to match with each value in *scalar-expression*.

See Also

- SELECT statement, HAVING clause, WHERE clause

- Overview of Predicates

- Collation

EXISTS (SQL)

Checks for the existence of a given object.

Synopsis

Checking a table for the existence of at least one row

```
EXISTS select-statement
```

Suppressing errors if the target of a DROP command does not exist

```
DROP-command IF EXISTS name
```

Description

The **EXISTS** predicate is used to test a specified table, typically for existence of at least a row. Since the **SELECT** statement following the **EXISTS** is being checked for containing something, the clause is often of the form:

```
EXISTS (SELECT... FROM... WHERE...)
```

where a typical statement might be:

SQL

```
SELECT name
    FROM Table_A
    WHERE EXISTS
    (SELECT *
        FROM Table_B
        WHERE Table_B.Number = Table_A.Number)
```

In this example, the predicate tests for the existence of one or more rows specified by the subquery.

Note that the test must occur on a **SELECT** statement (not on a **UNION**).

The **NOT EXISTS** clause tests for the non-existence of a row in a table, as shown in the following example:

SQL

```
SELECT EmployeeName,Age
    FROM Employees
    WHERE NOT EXISTS (SELECT * FROM BonusTable
    WHERE NOT (BonusTable.Result = 'Positive'
    AND Employees.EmployeeNum = BonusTable.EmployeeNum))
```

EXISTS can be used wherever a predicate condition can be specified, as described in Overview of Predicates.

Where applicable, the system automatically applies Set-Valued Subquery Optimization (SVSO) to an **EXISTS** or **NOT EXISTS** subquery. For details on this optimization, and using the %NOSVSO keyword to override it, refer to "Query Optimization Options" on the FROM clause reference page.

The variation **IF EXISTS** can be used to condition the execution of a **DROP** command (such as **DROP TABLE**) on the existence of its target, as in the following statement:

SQL

```
DROP TABLE IF EXISTS Records
```

In this example, no error will occur if the table *Records* does not exist. The statement will return SQLCODE 1 and a message. This behavior takes priority to settings which govern DDL statements in the Management Portal or the configuration parameter file (CPF), which suppress the error silently.

Similarly, **IF NOT EXISTS** can be specified when using the command **CREATE TABLE**, as in the following statement:

```
CREATE TABLE IF NOT EXISTS Records (...)
```

In this example, if a *Records* table already exists, the command will do nothing. No error will occur, and the statement will return SQLCODE 1 and a message. This behavior takes priority over settings which govern DDL statements in the Management Portal or the configuration parameter file (CPF), which effectively overwrite the existing table and suppress the error silently. For further details, consult the section on methods to check for existing tables on the reference page.

Arguments

select-statement

A simple query, usually containing a condition expression.

DROP-command

One of the following commands: **DROP AGGREGATE**, **DROP DATABASE**, **DROP FUNCTION**, **DROP INDEX**, **DROP METHOD**, **DROP PROCEDURE**, **DROP QUERY**, **DROP ROLE**, **DROP TABLE**, **DROP TRIGGER**, **DROP USER**, **DROP VIEW**.

See Also

- SELECT statement, HAVING clause, WHERE clause

- Overview of Predicates

%FIND (SQL)

Matches a value to a set of generated values with bitmap chunks iteration.

Synopsis

```
scalar-expression %FIND valueset [SIZE ((nn))]
```

Description

The **%FIND** predicate allows you to filter a result set by selecting those data values that match the values specified in *valueset*, iterating through values in a sequence of bitmap chunks. This match is successful when a *scalar-expression* value matches a value in *valueset*. If the *valueset* values do not match any of the scalar expression values, **%FIND** returns the null string. This match is always performed on the logical (internal storage) data value, regardless of the display mode.

%FIND, like the other comparison conditions, is used in the **WHERE** clause or the **HAVING** clause of a **SELECT** statement.

%FIND enables filtering of field values using an abstract, programmatically specified set of matching values. Specifically, it enables filtering of RowId field values using an abstract, programmatically specified bitmap, where *valueset* behaves similar to the subscript layer of a bitmap index.

The user-defined class is derived from the abstract class %SQL.AbstractFind. this abstract class defines the **ContainsItem()** boolean method. The **ContainsItem()** method matches the *scalar-expression* values to the *valueset* values.

Iteration through values in a sequence of bitmap chunks is performed using the following three methods:

- **GetChunk(c)**, which returns the bitmap chunk with chunk number *c*.

- **NextChunk(.c)**, which returns the first bitmap chunk with chunk number $> c$.

- **PreviousChunk(.c)**, which returns the first bitmap chunk with chunk number $< c$.

Refer to %SQL.AbstractFind for further details concerning these four methods.

Collation Types

%FIND uses the same collation type as the column it is matched against. By default, string data type fields are defined with SQLUPPER collation, which is not case-sensitive. You can define the string collation default for the current namespace and specify a non-default field collation type when defining a field/property. If you assign a different collation type to the column, you must also apply this collation type to the **%FIND** *substring*. Refer to %SQLUPPER for further information on case transformation functions.

SIZE Clause

The optional **%FIND** SIZE clause provides the integer *nn*, which specifies an order-of-magnitude estimate of the number of values in *valueset*. InterSystems IRIS uses this order-of-magnitude estimate to determine the optimal query plan. Specify *nn* as one of the following literals: 10, 100, 1000, 10000, etc. Because *nn* must be available as a constant value at compile time, it must be specified as a literal in all SQL code. Note that nesting parentheses must be specified as shown for all SQL, with the exception of Embedded SQL.

%FIND and %INSET Compared

- **%INSET** is the simplest and most general interface. It supports the **ContainsItem()** method.

- **%FIND** supports iteration over bitmap chunks using a bitmap index. It emulates the functionality of the ObjectScript $ORDER function, supporting **NextChunk()**, **PreviousChunk()**, and **GetChunk()** iteration methods, as well as the **ContainsItem()** method.

Arguments

scalar-expression

A scalar expression (most commonly the RowId field of a table) whose values are being compared with *valueset*.

valueset

An object reference (OREF) to a user-defined object that implements bitmap chunks iteration methods and the **ContainsItem()** method. This method takes a set of data values and returns a boolean when there is a match with a value in *scalar-expression*.

SIZE ((nn))

An optional order-of-magnitude integer (10, 100, 1000, etc.) used for query optimization.

See Also

- SELECT statement, HAVING clause, WHERE clause

- %INSET predicate

- Overview of Predicates

- SEARCH_INDEX function

FOR SOME (SQL)

Determines whether to return a record based on a condition test of field values.

Synopsis

```
FOR SOME (table [AS t-alias]) (fieldcondition)
```

Description

The **FOR SOME** predicate allows you to determine whether to return a record based on a boolean condition test of the values of one or more fields in a table. If *fieldcondition* evaluates as true, the record is returned. If *fieldcondition* evaluates as false, the record is not returned.

FOR SOME can be used wherever a predicate condition can be specified, as described in Overview of Predicates.

Delimiting parentheses are mandatory for the *table* (and its optional *t-alias*) argument. Delimiting parentheses are also mandatory for the *fieldcondition* argument. Whitespace is permitted, but not required, between these two sets of parentheses.

Commonly, **FOR SOME** is used to determine whether to return a record from a table based on the contents of a record in another table. **FOR SOME** can also be used to determine whether to return a record from a table based on the contents of a record in the same table. In this latter case, either all records are returned or no records are returned.

Compound Conditions

A *fieldcondition* can contain more than one condition expression. The set of conditions is enclosed in parentheses. Multiple conditions are specified with the logical operators AND and OR, which can also be specified using the & and ! symbols. A logical operator may be followed by the NOT unary operator. By default, conditions are evaluated in left-to-right order. You can specify a different order of evaluation by grouping multiple conditions using parentheses.

SQL

```
SELECT Name,COUNT(Name) AS NameCount
FROM Sample.Person AS p
WHERE FOR SOME (Sample.Employee AS e)(e.Name=p.Name AND p.Name %STARTSWITH 'A')
ORDER BY Name
```

SQL

```
SELECT Name,COUNT(Name) AS NameCount
FROM Sample.Person AS p
WHERE FOR SOME (Sample.Employee AS e)(e.Name=p.Name OR  p.Name %STARTSWITH 'A')
ORDER BY Name
```

Multiple Tables

You can specify multiple tables as a comma-separated list before the *fieldcondition*. The condition that determines whether to return records may reference the table from which data is being selected, or may reference field values in another table. Table aliases are usually required to associate each specified field with its table.

Arguments

table

table can be a single table or a comma-separated list of tables. The enclosing parentheses are mandatory.

AS t-alias

An optional alias for the preceding *table* name. An alias must be a valid identifier; it can be a delimited identifier.

fieldcondition

fieldcondition specifies one or more condition expressions referencing one or more fields. The *fieldcondition* is enclosed with parentheses. You can specify multiple condition expressions within *fieldcondition* using AND (&) and OR (!) logical operators. A subquery, enclosed in parentheses, which returns a result set from a single column that is used for the comparison with *scalar-expression*.

Examples

In the following example, **FOR SOME** returns all records in the Sample.Person table in which its Name field value matches the Name field value in the Sample.Employee table:

SQL

```
SELECT Name,COUNT(Name) AS NameCount
FROM Sample.Person AS p
WHERE FOR SOME (Sample.Employee AS e)(e.Name=p.Name)
ORDER BY Name
```

In the following example, **FOR SOME** returns records in the Sample.Person table based on a boolean test of the *same* table. This program returns all Sample.Person records if at least one record has an Age value greater than 65. Otherwise, it returns no records. Because at least one record in Sample.Person has an Age field value greater than 65, all Sample.Person records are returned:

SQL

```
SELECT Name,Age,COUNT(Name) AS NameCount
FROM Sample.Person
WHERE FOR SOME (Sample.Person)(Age>65)
ORDER BY Age
```

Like most predicates, **FOR SOME** can be inverted using the NOT logical operator, as shown in the following example:

SQL

```
SELECT Name,Age,COUNT(Name) AS NameCount
FROM Sample.Person
WHERE NOT FOR SOME (Sample.Person)(Age>65)
ORDER BY Age
```

In the following example, **FOR SOME** returns all records in the Sample.Person table in which its Name field value matches the Name field value in the Sample.Employee table, and their residence (Home_State) is in the same state as their office (Office_State):

SQL

```
SELECT Name,Home_State,COUNT(Name) AS NameCount
FROM Sample.Person AS p
WHERE FOR SOME (Sample.Employee AS e)(p.Name=e.Name AND p.Home_State=e.Office_State)
ORDER BY Name
```

In the following example, all records are returned if there is at least one Name in the Sample.Person table that is also found in the Sample.Employee table. Because this condition is true for at least one record, all Sample.Person records are returned:

SQL

```
SELECT Name AS PersonName,Age,COUNT(Name) AS NameCount
FROM Sample.Person
WHERE FOR SOME (Sample.Employee AS e,Sample.Person AS p) (e.Name=p.Name)
ORDER BY Name
```

In the following example, all records are returned if there is at least one Name in the Sample.Person table that is also found in the Sample.Company table. Because names of persons and names of companies (in this data set) are never the same, this condition is not true for any record. Therefore, no Sample.Person records are returned:

SQL

```
SELECT Name AS PersonName,Age,COUNT(Name) AS NameCount
FROM Sample.Person
WHERE FOR SOME (Sample.Company AS c,Sample.Person AS p) (c.Name=p.Name)
ORDER BY Name
```

See Also

- SELECT statement, HAVING clause, WHERE clause

- Overview of Predicates

- FOR SOME %ELEMENT predicate

FOR SOME %ELEMENT (SQL)

Matches list element values or the number of list elements with a predicate.

Synopsis

```
FOR SOME %ELEMENT(field) [[AS] e-alias] (predicate)
```

Description

The **FOR SOME %ELEMENT** predicate matches the list elements in *field* with the specified *predicate*. The SOME keyword specifies that at least one of the elements in the *field* must satisfy the specified *predicate* clause.

The *predicate* clause must contain either the %VALUE or the %KEY keyword, followed by a predicate condition. These keywords are not case-sensitive.

The use of %VALUE and %KEY is explained in the following examples:

- (%VALUE='Red') matches all *field* values that contain the value Red as one of their list elements. The *field* may only contain the single element Red, or it may contain multiple elements, one of which is the element Red.

- (%KEY=2) matches all *field* values that contain at least 2 elements. The *field* may contain exactly two elements or it may contain more than two elements. The %KEY value must be a positive integer. (%KEY=0) does not match any *field* values.

FOR SOME %ELEMENT cannot be used to match a *field* that is NULL.

The *predicate* clause can use any predicate condition, not just the equality condition. The following are some examples of *predicate* clauses:

```
(%VALUE='Red')
(%VALUE > 21)
(%VALUE %STARTSWITH 'R')
(%VALUE [ 'e')
(%VALUE IN ('Red','Blue')
(%VALUE IS NOT NULL)
(%KEY=3)
(%KEY > 1)
(%KEY IS NOT NULL)
```

Note: When supplying the predicate value at runtime (using a ? input parameter or a :var input host variable), the resulting predicate %STARTSWITH 'abc' gives better performance than the equivalent resulting predicate LIKE 'abc%'.

You can specify multiple predicate conditions using AND, OR, and NOT logical operators. InterSystems IRIS applies the combined predicate conditions to each element. Therefore, it is not meaningful to apply two %VALUE or two %KEY predicates using an AND test.

For example, using **FOR SOME %ELEMENT** to match a field containing the values Red, Green, Red Green, Black Red, Green Yellow Red, Green Black, Yellow, or Black Yellow:

- (%VALUE='Red') matches any field containing the element Red: Red, Red Green, Black Red, and Red Yellow Green.

- (%VALUE='Red' OR %VALUE='Green') matches any field containing either element (or both, in any order): Red, Green, Red Green, Black Red, Green Yellow Red, Green Black. This is functionally identical to (%VALUE IN('Red','Green')).

- (%VALUE='Red' AND %VALUE='Green') matches no field values because it matches each element against both Red and Green, and no element can have the value Red and the value Green. This predicate does *not* match the two-element value Red Green.

- `(%VALUE='Red' AND %KEY=2)` matches Red Green, Black Red, Green Yellow Red.

- `(%VALUE='Red' OR %KEY=2)` matches Red, Red Green, Black Red, Green Yellow Red, Green Black, Black Yellow.

FOR SOME %ELEMENT is a collection predicate. It can be used in most contexts where a predicate condition can be specified, as described in Overview of Predicates. It is subject to the following restrictions:

- You cannot use **FOR SOME %ELEMENT** in a HAVING clause.

- You cannot use **FOR SOME %ELEMENT** as a predicate that selects fields for a **JOIN** operation.

- You cannot associate **FOR SOME %ELEMENT** with another predicate condition using the OR logical operator if the two predicates reference fields in different tables. For example:

```
WHERE FOR SOME %ELEMENT(t1.FavoriteColors) (%VALUE='purple') OR t2.Age < 65
```

Because this restriction depends on how the optimizer uses indexes, SQL may only enforce this restriction when indexes are added to a table. It is strongly suggested that this type of logic be avoided in all queries.

- You cannot use **FOR SOME %ELEMENT** when querying a sharded table. See Querying the Sharded Cluster.

Collection Index

An important use of **FOR SOME %ELEMENT** is to select elements using a collection index. If the appropriate KEYS or ELEMENTS index is defined for *field*, InterSystems IRIS uses this index rather than directly referencing the field value elements.

If the following collection index is defined:

Class Member

```
 INDEX fcIDX1 ON FavoriteColors(ELEMENTS);
```

The following query uses this index:

SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FOR SOME %ELEMENT(FavoriteColors) (%VALUE='Red')
```

If the following collection index is defined:

Class Member

```
 INDEX fcIDX2 ON FavoriteColors(KEYS) [ Type = bitmap ];
```

The following query uses this index:

SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FOR SOME %ELEMENT(FavoriteColors) (%KEY=2)
```

For further details on **FOR SOME %ELEMENT** with collection indexes, refer to Querying Collections.

Arguments

field

A scalar expression (most commonly a data column) whose elements are being compared with *predicate*.

AS *e-alias*

An optional element alias used to qualify %KEY or %VALUE within the *predicate*. Commonly, this alias is used when the *predicate* contains a nested **FOR SOME %ELEMENT** condition. The alias must be a valid identifier. The AS keyword is optional.

(predicate)

A predicate condition, enclosed in parentheses. Within this condition use %VALUE and/or %KEY to determine what the condition is matching. %VALUE matches the element value (%VALUE='Red'). %KEY matches the minimum number of elements (%KEY=2). Within this condition, %VALUE and %KEY may be optionally qualified if you have specified an *e-alias*. This predicate can consist of multiple condition expressions with AND and OR logical operators.

Examples

The following example uses **FOR SOME %ELEMENT** to return those rows where the FavoriteColors list contains the element 'Red':

SQL

```
SELECT Name,FavoriteColors
FROM Sample.Person
WHERE FOR SOME %ELEMENT(FavoriteColors) (%VALUE='Red')
```

In the following example, the %VALUE predicate contains an IN statement specifying a comma-separated list. This example returns those rows where the FavoriteColors list contains either the element 'Red' or the element 'Blue' (or both):

SQL

```
SELECT Name,FavoriteColors
FROM Sample.Person
WHERE FOR SOME %ELEMENT(FavoriteColors) (%VALUE IN ('Red','Blue'))
```

The following example uses a predicate clause with two Contains operators ([). It returns those rows where the Favorite-Colors list has an element that contains a lowercase 'l' and a lowercase 'e' (the contains operator is case-sensitive). In this case, the elements 'Blue', 'Yellow', and 'Purple':

SQL

```
SELECT Name,FavoriteColors AS Preferences
FROM Sample.Person
WHERE FOR SOME %ELEMENT(FavoriteColors) AS fc (fc.%VALUE [ 'l' AND fc.%VALUE [ 'e')
```

This example also demonstrates how an element alias (*e-alias*) is used.

The following Dynamic SQL example uses %KEY to return rows based on the number of elements in FavoriteColors. The first **%Execute()** sets %KEY=1, returning all rows that have one or more FavoriteColors elements. The second **%Execute()** sets %KEY=2, returning all rows that have two or more FavoriteColors elements:

ObjectScript

```
  SET q1 = "SELECT %ID,Name,FavoriteColors FROM Sample.Person "
  SET q2 = "WHERE FOR SOME %ELEMENT(FavoriteColors) (%KEY=?)"
  SET myquery = q1_q2
  SET tStatement = ##class(%SQL.Statement).%New()
  SET tStatement.%SelectMode=1
  SET qStatus = tStatement.%Prepare(myquery)
    IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET rset = tStatement.%Execute(1)
  DO rset.%Display()
  WRITE !,"End of data %KEY 1",!!
  SET rset = tStatement.%Execute(2)
  DO rset.%Display()
  WRITE !,"End of data %KEY 2"
```

See Also

- **SELECT** statement, **WHERE** clause
- **Overview of Predicates**
- **FOR SOME** predicate

IN (SQL)

Matches a value to items in an unstructured comma-separated list.

Synopsis

```
scalar-expression IN (item1,item2[,...])

scalar-expression IN (subquery)
```

Description

The **IN** predicate is used for matching a value to an unstructured series of items. Typically, it compares column data values to a comma-separated list of values. **IN** can perform equality comparisons and subquery comparisons.

Like most predicates, **IN** can be inverted using the NOT logical operator. Neither **IN** nor **NOT IN** can be used to return NULL fields. To return NULL fields use IS NULL.

IN can be used wherever a predicate condition can be specified, as described in Overview of Predicates.

Equality Comparison

IN comparisons use the collation type defined for the *scalar-expression*, regardless of the collation type of the individual *item*s. By default, string data type fields are defined with SQLUPPER collation, which is not case-sensitive. You can define the string collation default for the current namespace and specify a non-default field collation type when defining a field/property.

When dates or times are used for **IN** predicate equality comparisons, the appropriate data type conversions are automatically performed. If the WHERE field is type TimeStamp, values of type Date or Time are converted to Timestamp. If the WHERE field is type Date, values of type TimeStamp or String are converted to Date. If the WHERE field is type Time, values of type TimeStamp or String are converted to Time.

The following examples both perform the same equality comparisons and return the same data. The DOB field is of data type Date:

SQL

```
SELECT Name,DOB FROM Sample.Person
WHERE DOB IN ({d '1951-02-02'},{d '1987-02-28'})
```

SQL

```
SELECT Name,DOB FROM Sample.Person
WHERE DOB IN ({ts '1951-02-02 02:37:00'},{ts '1987-02-28 16:58:10'})
```

For further details refer to Date and Time Constructs.

%SelectMode

If %SelectMode is set to a value other than Logical format, the **IN** predicate values must be specified in the %SelectMode format (ODBC or Display). This applies mainly to dates, times, and InterSystems IRIS format lists (%List). Specifying predicate values in Logical format commonly results in an SQLCODE error. For example, SQLCODE -146 "Unable to convert date input to a valid logical date value".

In the following Dynamic SQL example, the **IN** predicate must specify dates in %SelectMode=1 (ODBC) format:

ObjectScript

```
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE DOB IN('1956-03-05','1956-04-08','1956-04-18','1956-12-08')"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

Subquery Comparison

You can use the **IN** predicate with a subquery to test whether a column value (or any other expression) equals any of the subquery row values. For example:

SQL

```
SELECT Name,Home_State FROM Sample.Person
WHERE Name IN
   (SELECT Name FROM Sample.Employee
    HAVING Salary < 50000)
```

Note that the subquery must have exactly one select-item in the SELECT list.

An **IN** cannot specify both a subquery and a comma-separated list of literal values.

Literal Substitution Override

You can override literal substitution during compile pre-parsing by enclosing each **IN** predicate argument with parentheses. For example, `WHERE Home_State IN (('ME'),('NH'),('VT'),('MA'),('RI'),('CT'))`. This may improve query performance by improving overall selectivity and/or subscript bounding selectivity. However, it should be avoided when the same query is called multiple times with different values, as it will result in the creation of a separate cached query for each query call.

IN and %INLIST

Both the **IN** and **%INLIST** predicates can be used to supply multiple values to use for OR equality comparisons. The **%INLIST** predicate is used for matching a value to the elements of a %List structure. In Dynamic SQL you can supply the **%INLIST** predicate values as a single host variable. You must supply the **IN** predicate values as individual host variables. Therefore, changing the number of **IN** predicate values results in the creation of a separate cached query. **%INLIST** takes a single predicate value, a %List with multiple elements; changing the number of %List elements does not result in the creation of a separate cached query. **%INLIST** also provides an order-of-magnitude SIZE argument that SQL uses to optimize performance. For these reasons it is often advantageous to use `%INLIST($LISTFROMSTRING(val))` rather than `IN(val1,val2,val3,..valn)`.

%INLIST can perform equality comparisons; it cannot perform subquery comparisons.

For further details, refer to %INLIST.

Arguments

scalar-expression

A scalar expression (most commonly a data column) whose values are being compared with a comma-separated list of values or the result set generated by the *subquery*.

item

One or more literal values, input host variables, or expressions that resolve to a literal value. List in any order, separate with commas.

subquery

A subquery, enclosed in parentheses, which returns a result set from a single column that is used for the comparison with *scalar-expression*.

Examples

The **IN** predicate can serve as shorthand for the use of multiple equality comparisons linked together with the OR operator. For instance:

SQL

```
SELECT Name, Home_State FROM Sample.Person
WHERE Home_State IN ('ME','NH','VT','MA','RI','CT')
```

evaluates true if Home_State equals any of the values in the comma-separated list. The listed items can be constants or expressions.

The following two examples show that collation matching is based on the *scalar-expression* collation. The Home_State field is defined with SQLUPPER (not case-sensitive) collation. Therefore, the following example returns NH Home_State values:

SQL

```
SELECT Name, Home_State FROM Sample.Person
WHERE Home_State IN ('ME','nH','VT')
```

The following example does not return NH Home_State values:

SQL

```
SELECT Name, Home_State FROM Sample.Person
WHERE %EXACT(Home_State) IN ('ME','nH','VT')
```

It is not meaningful to include NULL in the list of values. NULL is the absence of a value, and therefore fails all equality tests. Specifying an **IN** predicate (or any other predicate) eliminates any instances of the specified field that are NULL. This is shown in the following *incorrect* (but executable) example:

SQL

```
SELECT FavoriteColors FROM Sample.Person
WHERE FavoriteColors IN ($LISTBUILD('Red'),$LISTBUILD('Blue'),NULL)
  /* NULL here is meaningless. No FavoriteColor NULL fields returned */
```

The only way to include a field with NULL in the predicate result set is to specify the IS NULL predicate, as shown in the following example:

SQL

```
SELECT FavoriteColors FROM Sample.Person
WHERE FavoriteColors IN ($LISTBUILD('Red'),$LISTBUILD('Blue')) OR FavoriteColors IS NULL
```

The following example uses an IN subquery to return the Employee states that are not Vendor states:

SQL

```
SELECT Home_State
FROM Sample.Employee
WHERE Home_State NOT IN (SELECT Address_State FROM Sample.Vendor)
GROUP BY Home_State
```

The following example matches a collation function expression to an **IN** predicate with a subquery:

SQL

```
SELECT Name,Id FROM Sample.Person
WHERE %EXACT(Spouse) NOT IN
    (SELECT Id FROM Sample.Person
     WHERE Age < 65)
```

See Also

- SELECT statement, HAVING clause, WHERE clause

- %INLIST predicate

- Overview of Predicates

%INLIST (SQL)

Matches a value to the elements in a %List structured list.

Synopsis

```
scalar-expression %INLIST list [SIZE ((nn))]
```

Arguments

Argument	Description
scalar-expression	A scalar expression (most commonly a data column) whose values are being compared with *list* elements.
list	A %List structure containing one or more elements.
SIZE ((*nn*))	*Optional* — An integer specifying an order-of-magnitude estimate of the number of elements in *list*. Must be specified as a literal with one of the following values: 10, 100, 1000, 10000, and so forth.

Description

The **%INLIST** predicate is an InterSystems IRIS extension for matching the values of a field with the elements of a list structure. Both **%INLIST** and **IN** allow you to perform such equality comparisons with multiple specified values. **%INLIST** specifies these multiple values as the elements of a single *list* argument. Therefore, **%INLIST** allows you to vary the number of values to match without creating a separate cached query.

The optional **%INLIST** SIZE clause provides the integer *nn*, which specifies an order-of-magnitude estimate of the number of list elements in *list*. InterSystems IRIS uses this order-of-magnitude estimate to determine the optimal query plan. Because the same cached query is used regardless of the number of elements in *list*, specifying SIZE allows you to create a cached query optimized for the anticipated approximate number of elements in *list*. Changing the SIZE literal creates a separate cached query. Specify *nn* as one of the following literals: 10, 100, 1000, 10000, etc. Because *nn* must be available as a constant value at compile time, it must be specified as a literal in all SQL code. Note that double parentheses must be specified as shown for all compiled SQL (Dynamic SQL). Double parentheses are not used with Embedded SQL.

%INLIST performs an equality comparison with each of the elements of *list*. **%INLIST** comparisons use the collation type defined for the *scalar-expression*. Therefore, comparisons of *list* elements may be case-sensitive or not case-sensitive, depending on the collation of *scalar-expression*. By default, string data type fields are defined with SQLUPPER collation, which is not case-sensitive. You can define the string collation default for the current namespace and specify a non-default field collation type when defining a field/property.

It is not meaningful to specify NULL as a comparison value. NULL is the absence of a value, and therefore fails all equality tests. Specifying an **%INLIST** predicate (or any other predicate) eliminates any instances of the specified field that are NULL. You must specify the **IS NULL** predicate to include fields with NULL in the predicate result set.

Like most predicates, **%INLIST** can be inverted using the NOT logical operator. Neither **%INLIST** nor **NOT %INLIST** can be used to return NULL fields. To return NULL fields use IS NULL.

If the match expression is not in %List format, **%INLIST** generates an SQLCODE -400 error. For example, if the SqlListType of the collection property is DELIMITED, the logical value of the list field is not in %List format. For further details on list structures, see the SQL $LIST function.

%INLIST can be used wherever a predicate condition can be specified, as described in Overview of Predicates.

For matching a value to an unstructured series of items, such as a comma-separated list of values, use the IN predicate. **IN** can perform equality comparisons and subquery comparisons.

%SelectMode

The **%INLIST** predicate does not use the current %SelectMode setting. The elements of *list* should be specified in Logical format, regardless of the %SelectMode setting. Attempting to specify *list* elements in ODBC format or Display format commonly results in no data matches or unintended data matches.

You can use the %EXTERNAL or %ODBCOUT format-transform functions to transform the *scalar-expression* field that the predicate operates upon. This allows you to specify the *list* elements in Display format or ODBC format. However, using a format-transform function prevents the use of the index for the field, and can thus have a significant performance impact.

In the following Dynamic SQL example, the **%INLIST** predicate specifies a list containing date value elements for the year 1978 in Logical format, not in %SelectMode=1 (ODBC) format. Dates that correspond to these $HOROLOG format dates are selected:

ObjectScript

```
SET bday=$LISTBUILD(50039)
FOR i=50039:1:50403 {SET bday=bday_$LISTBUILD(i) }
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE DOB %INLIST ?"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(bday)
DO rset.%Display()
```

The following Dynamic SQL example uses the %ODBCOUT format-transform function to transform the DOB field matched by the predicate. This allows you to specify the **%INLIST** list elements in ODBC format. However, specifying the format-transform function prevents the use of an index for DOB field values:

ObjectScript

```
SET births=$LISTBUILD("1978-01-15","1978-08-22","1978-10-01")
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE %ODBCOUT(DOB) %INLIST ?"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(births)
DO rset.%Display()
```

%INLIST and IN

Both the **%INLIST** and **IN** predicates can be used to supply multiple values to use for equality comparisons. The following examples return the same results:

SQL

```
SELECT Name, Home_State
FROM Sample.Person
WHERE Home_State %INLIST $LISTBUILD('VT','NH','ME')
```

SQL

```
SELECT Name,Home_State
FROM Sample.Person
WHERE Home_State IN('VT','NH','ME')
```

For Dynamic SQL, you can supply the **%INLIST** predicate values as a single host variable; you must supply the **IN** predicate values as individual host variables. Therefore, changing the number of **IN** predicate values results in the creation

of a separate cached query. Changing the number of **%INLIST** predicate values does not result in the creation of a separate cached query. For further details, refer to Cached Queries.

Examples

The following two examples show that collation matching is based on the *scalar-expression* collation. The Home_State field is defined with SQLUPPER collation which is not case-sensitive. The *list* in these examples specifies New Hampshire as "nH", rather than "NH". The first example returns NH Home_State values, the second example does not return NH Home_State values:

SQL

```
SELECT Name,Home_State
FROM Sample.Person
WHERE Home_State %INLIST $LISTBUILD("VT","nH","ME")
```

SQL

```
SELECT Name,Home_State
FROM Sample.Person
WHERE %EXACT(Home_State) %INLIST $LISTBUILD("VT","nH","ME")
```

The following example creates a cached query with a SIZE literal of 10. Specifying SIZE 10 is optimal for this query, because 10 corresponds in order-of-magnitude to the actual number of elements in the list. Changing the number of elements in the list does not create a separate cached query. Changing the SIZE literal does create a separate cached query:

SQL

```
SELECT Name,Home_State
FROM Sample.Person
WHERE Home_State %INLIST $LISTBUILD("VT","NH","ME") SIZE ((10))
```

See Also

- SELECT statement, HAVING clause, WHERE clause

- $LISTBUILD function

- IN predicate

- Overview of Predicates

%INSET (SQL)

Matches a value to a set of generated values.

Synopsis

```
scalar-expression %INSET valueset [SIZE ((nn))]
```

Description

The **%INSET** predicate allows you to filter a result set by selecting those data values that match the values specified in *valueset*. This match is successful when a *scalar-expression* value matches a value in *valueset*. If the *valueset* values do not match any of the scalar expression values, **%INSET** returns the null string. This match is always performed on the logical (internal storage) data value, regardless of the display mode.

%INSET is never true for a NULL value. Therefore, it will not match a NULL in the *scalar-expression* with a NULL in *valueset*.

%INSET, like the other comparison conditions, is used in the **WHERE** clause or the **HAVING** clause of a **SELECT** statement.

%INSET enables filtering of field values using an abstract, programmatically specified set of matching values. Specifically, it enables filtering of RowId field values using an abstract, programmatically specified temp-file or bitmap index, where *valueset* behaves similar to the lowest subscript layer of a bitmap index or a regular index.

The user-defined class is derived from the abstract class %SQL.AbstractFind. this abstract class defines the **ContainsItem()** method, which is the only method supported by **%INSET**. The **ContainsItem()** method returns the *valueset*. Refer to %SQL.AbstractFind for further details.

Collation Types

%INSET uses the same collation type as the column it is matched against. By default, string data type fields are defined with SQLUPPER collation, which is not case-sensitive. You can define the string collation default for the current namespace and specify a non-default field collation type when defining a field/property. If you assign a different collation type to the column, you must also apply this collation type to the **%INSET** *substring*. Refer to %SQLUPPER for further information on case transformation functions.

SIZE Clause

The optional **%INSET** SIZE clause provides the integer *nn*, which specifies an order-of-magnitude estimate of the number of values in *valueset*. InterSystems IRIS uses this order-of-magnitude estimate to determine the optimal query plan. Specify *nn* as one of the following literals: 10, 100, 1000, 10000, etc. Because *nn* must be available as a constant value at compile time, it must be specified as a literal in all SQL code. Note that nesting parentheses must be specified as shown for all SQL, with the exception of Embedded SQL.

%INSET and %FIND Compared

- **%INSET** is the simplest and most general interface. It supports the **ContainsItem()** method.

- **%FIND** supports iteration over bitmap chunks using a bitmap index. It emulates the functionality of the ObjectScript $ORDER function, supporting **NextChunk()**, **PreviousChunk()**, and **GetChunk()** iteration methods, as well as the **ContainsItem()** method.

-

Arguments

scalar-expression

A scalar expression (most commonly the RowId field of a table) whose values are being compared with *valueset*.

valueset

An object reference (OREF) to a user-defined object that implements a **ContainsItem()** method. This method takes a set of data values and returns a boolean when there is a match with a value in *scalar-expression*.

SIZE ((nn))

An optional order-of-magnitude integer (10, 100, 1000, etc.) used for query optimization.

See Also

- SELECT statement, HAVING clause, WHERE clause

- %FIND predicate

- Overview of Predicates

IS JSON (SQL)

Determines if a data value is in JSON format.

Synopsis

```
scalar-expression IS [NOT] JSON [keyword]
```

Description

The **IS JSON** predicate determines if a data value is in JSON format.

IS JSON (with or without the optional VALUE keyword) returns true for any JSON array or JSON object. This includes an empty JSON array `'[]'` or an empty JSON object `'{}'`.

The VALUE keyword and the SCALAR keyword are synonyms.

For further details, refer to the ObjectScript **SET** command subsection "JSON Object and JSON Array".

The **IS NOT JSON** predicate is one of the few predicates that can be used on a stream field in a **WHERE** clause. Its behavior is the same as **IS NOT NULL**.

IS JSON can be used wherever a predicate condition can be specified, as described in Overview of Predicates.

Arguments

scalar-expression

A scalar expression that is being checked for JSON formatting.

keyword

An optional argument. One of the following: VALUE, SCALAR, ARRAY, or OBJECT. The default is VALUE.

Examples

The following example determines if the predicate is a properly-formatted JSON string, either a JSON object or a JSON array:

SQL

```
SELECT TOP 5 Name FROM Sample.Person
    WHERE '{""name"":""Fred"",""spouse"":""Wilma""}' IS JSON
```

IS JSON ARRAY returns true for a JSON array OREF. **IS JSON OBJECT** returns true for a JSON object OREF. This is shown in the following examples:

SQL

```
  SET jarray=
  WRITE "JSON array: ",jarray,!
  SET myquery = "SELECT TOP 5 Name FROM Sample.Person WHERE ? IS JSON ARRAY"
  SET tStatement = ##class(%SQL.Statement).%New()
  SET qStatus = tStatement.%Prepare(myquery)
   IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET rset = tStatement.%Execute(jarray)
  DO rset.%Display()
```

ObjectScript

```
SET jarray=[1,2,3,5,8,13,21,34]
WRITE "JSON array: ",jarray,!
SET myquery = "SELECT TOP 5 Name FROM Sample.Person WHERE ? IS JSON OBJECT"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
 IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(jarray)
DO rset.%Display()
```

ObjectScript

```
SET jobj={"name":"Fred","spouse":"Wilma"}
WRITE "JSON object: ",jobj,!
SET myquery = "SELECT TOP 5 Name FROM Sample.Person WHERE ? IS JSON OBJECT"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
 IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(jobj)
DO rset.%Display()
```

The following example shows the behavior of the **IS NOT JSON** predicate:

SQL

```
SELECT Title,%OBJECT(Picture) AS PhotoOref FROM Sample.Employee
    WHERE Picture IS NOT JSON
```

See Also

- SELECT statement, HAVING clause, WHERE clause

- JSON_ARRAY, JSON_OBJECT functions

- JSON_ARRAYAGG aggregate function

- Overview of Predicates

IS NULL (SQL)

Determines if a data value is NULL.

Synopsis

```
scalar-expression IS [NOT] NULL
```

Description

The **IS NULL** predicate detects undefined values. You can detect all null values, or all non-null values:

SQL

```
SELECT Name, FavoriteColors FROM Sample.Person
WHERE FavoriteColors IS NULL
```

SQL

```
SELECT Name, FavoriteColors FROM Sample.Person
WHERE FavoriteColors IS NOT NULL
```

The **IS NULL** / **IS NOT NULL** predicate is one of the few predicates that can be used on a stream field in a **WHERE** clause. This is shown in the following example:

SQL

```
SELECT Title,%OBJECT(Picture) AS PhotoOref FROM Sample.Employee
WHERE Picture IS NOT NULL
```

IS NULL can be used wherever a predicate condition can be specified, as described in Overview of Predicates.

The **IS NULL** predicate should not be confused with the SQL ISNULL function.

See Also

- SELECT statement, HAVING clause, WHERE clause

- Overview of Predicates

LIKE (SQL)

Matches a value with a pattern string containing literals and wildcards.

Synopsis

```
scalar-expression LIKE pattern [ESCAPE char]
```

Arguments

Argument	Description
scalar-expression	A scalar expression (most commonly a data column) whose values are being compared with *pattern*.
pattern	A quoted string representing the pattern of characters to match with each value in *scalar-expression*. The *pattern* string can contain literal characters, and the underscore (_) and percent (%) wildcard characters.
ESCAPE *char*	*Optional* — A string containing a single character. This *char* character can be used in *pattern* to specify that the character immediately following it is to be treated as a literal.

Description

The **LIKE** predicate allows you to select those data values that match the character or characters specified in *pattern*. The *pattern* may contain wildcard characters. If *pattern* does not match any of the scalar expression values, **LIKE** returns the null string.

LIKE can be used wherever a predicate condition can be specified, as described in Overview of Predicates.

The **LIKE** predicate supports the following wildcards:

Table D–1: LIKE Wildcard Characters

Character	Matches
_	Any single character.
%	Any sequence of 0 or more characters. (In accordance with the SQL standard, NULL is *not* considered a sequence of 0 characters, and is thus not selected by this wildcard.)

In Dynamic SQL or Embedded SQL, a *pattern* can represent wildcard characters and input parameters or input host variables as concatenated strings, as shown in the Examples section.

Note: When supplying the predicate value at runtime (using a ? input parameter or a :var input host variable), the resulting predicate %STARTSWITH 'abc' gives better performance than the equivalent resulting predicate LIKE 'abc%'.

Collation Types

The *pattern* string uses the same collation type as the column it is matching against. By default, string data type fields are defined with SQLUPPER collation, which is not case-sensitive. You can define the string collation default for the current namespace and specify a non-default field collation type when defining a field/property. If a query includes the ESCAPE *char* clause, the escaping occurs after collation.

If **LIKE** is applied against a field with the SQLUPPER default collation type, the **LIKE** clause returns matches that ignore letter case. You can use the SQLSTRING collation type to perform a **LIKE** string comparison that is case-sensitive.

The following example returns all names that contain the substring "Ro". Because **LIKE** is not case-sensitive, **LIKE '%Ro%'** returns Robert, Rogers, deRocca, LaRonga, Brown, Mastroni, and so forth:

SQL

```
SELECT Name FROM Sample.Person
WHERE Name LIKE '%Ro%'
```

Compare this to the Contains operator ([), which uses EXACT (case-sensitive) collation:

SQL

```
SELECT Name FROM Sample.Person
WHERE Name [ 'Ro'
```

By using the %SQLSTRING collation type, you can use **LIKE** to return only those names that contain the case-sensitive substring "Ro". It would not return Mastroni or Brown:

SQL

```
SELECT Name FROM Sample.Person
WHERE %SQLSTRING(Name) LIKE '%Ro%'
```

In the above example, the leading space that %SQLSTRING prepended to Name values was handled by the % wildcard. A more robust example would specify the collation type on both sides of the predicate:

SQL

```
SELECT Name FROM Sample.Person
WHERE %SQLSTRING(Name) LIKE %SQLSTRING('%Ro%')
```

Refer to %SQLUPPER for further information on case transformation functions.

All Values, Empty String Values, and NULL

If the *pattern* value is percent (%), **LIKE** selects all values for the specified field, including empty string values:

SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FavoriteColors LIKE '%'
```

It does not select fields that are NULL.

Specifying a *pattern* value of empty string returns empty string values.

SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FavoriteColors LIKE ''
```

Specifying a *pattern* value of NULL is not a meaningful operation. It completes successfully, but returns no values.

SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FavoriteColors LIKE NULL
```

Like most predicates, **LIKE** can be inverted using the NOT logical operator. Neither **LIKE** nor **NOT LIKE** can be used to return NULL fields. To return NULL fields use IS NULL.

ESCAPE Clause

ESCAPE permits the use of a wildcard character as a literal character within *pattern*. ESCAPE *char*, if provided and if it is a single character, indicates that any character directly following it in *pattern* is to be understood as a literal character, rather than a wildcard or formatting character. The following example shows the use of ESCAPE to return values that contain the string '_SYS':

SQL

```
SELECT * FROM MyTable
WHERE symbol_field LIKE '%\_SYS%' ESCAPE '\'
```

%SelectMode

The **LIKE** predicate does not use the current %SelectMode setting. A *pattern* should be specified in Logical format, regardless of the %SelectMode setting. Attempting to specify a *pattern* in ODBC format or Display format commonly results in no data matches or unintended data matches.

You can use the %EXTERNAL or %ODBCOUT format-transform functions to transform the *scalar-expression* field that the predicate operates upon. This allows you to specify the *pattern* in Display format or ODBC format. However, using a format-transform function prevents the use of the index for the field, and can thus have a significant performance impact.

In the following Dynamic SQL example, the **LIKE** predicate specifies the date *pattern* in Logical format, not in %SelectMode=1 (ODBC) format. Rows with DOB Logical values beginning with 41 (dates from April 4 1953 ($HOROLOG 41000) through December 28 1955 ($HOROLOG 41999)) are selected:

ObjectScript

```
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE DOB LIKE '41%'"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

The following example uses the %ODBCOUT format-transform function to transform the DOB field matched by the predicate. This allows you to specify the **LIKE** *pattern* in ODBC format. It selects rows with DOB field ODBC values beginning with 195 (dates within the range of years 1950 through 1959). However, specifying the format-transform function prevents the use of an index for DOB field values:

SQL

```
  SELECT Name,DOB FROM Sample.Person
WHERE %ODBCOUT(DOB) LIKE '195%'
```

Literal Substitution Override

You can override literal substitution during compile pre-parsing by enclosing the **LIKE** predicate argument with double parentheses. For example, WHERE Name LIKE (('Mc%')) or WHERE Name LIKE (('%son%')). This may improve query performance by improving overall selectivity and/or subscript bounding selectivity. However, it should be avoided when the same query is called multiple times with different values, as it will result in the creation of a separate cached query for each query call.

Examples

The following example uses the **WHERE** clause to select Name values that contain "son", including those that begin or end with "son". By default, **LIKE** string comparisons are not case-sensitive:

SQL

```
SELECT %ID,Name FROM Sample.Person
WHERE Name LIKE '%son%'
```

The following Embedded SQL example returns the same result set as the previous example. Note how the input host variable (:subname) is specified in the **LIKE** *pattern* using the concatenation operator:

ObjectScript

```
  SET subname="son"
  &sql(DECLARE C1 CURSOR FOR SELECT %ID,Name INTO :id,:nameout FROM Sample.Person
      WHERE Name LIKE '%'_:subname_'%')
  &sql(OPEN C1)
      QUIT:(SQLCODE'=0)
 &sql(FETCH C1)
 WHILE (SQLCODE = 0) {
     WRITE id," ",nameout,!
   &sql(FETCH C1) }
  &sql(CLOSE C1)
```

The following Dynamic SQL example returns the same result set as the previous example. Note how the input parameter (?) is specified in the **LIKE** *pattern* using the concatenation operator:

ObjectScript

```
  SET myquery = "SELECT %ID,Name FROM Sample.Person WHERE Name LIKE '%'_?_'%'"
  SET tStatement = ##class(%SQL.Statement).%New()
  SET qStatus = tStatement.%Prepare(myquery)
    IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET rset = tStatement.%Execute("son")
  DO rset.%Display()
```

The following example uses the **WHERE** clause to select FavoriteColors values that contain "blue". The FavoriteColors field is a %List field; the % wildcards handle the %List formatting characters:

SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FavoriteColors LIKE '%blue%'
```

The following example uses a **HAVING** clause to select records for people whose age starts with a 1 followed by a single character. It displays the average for all ages and the average for the ages selected by the **HAVING** clause. It orders the results by age. All returned values have ages from 10 through 19.

SQL

```
SELECT Name,
       Age,
       AVG(Age) AS AvgAge,
       AVG(Age %AFTERHAVING) AS AvgTeen
FROM Sample.Person
HAVING Age LIKE '1_'
ORDER BY Age
```

See Also

*   SELECT statement, HAVING clause, WHERE clause

*   %MATCHES predicate

*   %PATTERN predicate

*   %STARTSWITH predicate

*   Overview of Predicates

%MATCHES (SQL)

Matches a value with a pattern string containing literals, wildcards, and ranges.

Synopsis

```
scalar-expression %MATCHES pattern [ESCAPE char]
```

Description

The **%MATCHES** predicate is an InterSystems IRIS extension for matching a value to a pattern string. **%MATCHES** returns True or False for the match operation. The *pattern* string can consist of literal characters, wild card characters, and list or ranges of matching literals.

Pattern matches are case-sensitive. Pattern matching is based on the EXACT value of *scalar-expression*, not its collation value. Therefore, a **%MATCHES** operation is always case-sensitive, even when the collation type of *scalar-expression* is not case-sensitive.

%MATCHES supports the following *pattern* wildcards:

?	Matches any single character of any type.
*	Matches zero or more characters of any type.
[abc]	Matches any one of the characters specified in brackets.
[a-z]	Matches character within the range specified in brackets, inclusive of the specified characters.
[^A-Z] [^a-z] [^0–9]	These ranges match any characters *except* those specified in brackets. You can use this syntax to specify no uppercase letters, or no lowercase letters, or no numbers. Only the specified literal ranges shown are supported.
\	Treats the character following as a literal character, rather than as a wildcard. Backslash is the default escape character; you can specify another character as the escape character using the optional ESCAPE clause.

Like most predicates, **%MATCHES** can be inverted using the NOT operator: `item NOT %MATCHES pattern`. Neither **%MATCHES** nor **NOT %MATCHES** can be used to return NULL fields. To return NULL fields use IS NULL.

The backslash (\) character is the default escape character. It can be used to specify that a wildcard character is to be used as a literal match at the specified pattern location. For example, to match a question mark as the first character of a string specify `'\?*'`. To match a question mark as the fourth character of a string specify `'???\?*'`. To match a question mark anywhere in a string specify `'*\?*'`. To match a string that consists of only an asterisk character specify `'\*'`. To match a string that contains at least one asterisk character specify `'*\**'`. To match a backslash character anywhere in a string specify `'*\\*'`.

%MATCHES can be used wherever a predicate condition can be specified, as described in Overview of Predicates.

%MATCHES is supported for compatibility with Informix SQL.

%SelectMode

The **%MATCHES** predicate does not use the current %SelectMode setting. A *pattern* should be specified in Logical format, regardless of the %SelectMode setting. Attempting to specify a *pattern* in ODBC format or Display format commonly results in no data matches or unintended data matches.

You can use the %EXTERNAL or %ODBCOUT format-transform functions to transform the *scalar-expression* field that the predicate operates upon. This allows you to specify the *pattern* in Display format or ODBC format. However, using a format-transform function prevents the use of the index for the field, and can thus have a significant performance impact.

In the following Dynamic SQL example, the **%MATCHES** predicate specifies the date *pattern* in Logical format, not in %SelectMode=1 (ODBC) format. Rows with DOB Logical values beginning with 41 (dates from April 4 1953 ($HOROLOG 41000) through December 28 1955 ($HOROLOG 41999)) are selected:

ObjectScript

```
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE DOB %MATCHES '41*'"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

The following Dynamic SQL example uses the %ODBCOUT format-transform function to transform the DOB field matched by the predicate. This allows you to specify the **%MATCHES** *pattern* in ODBC format. It selects rows with DOB field ODBC values beginning with 195 (dates within the range of years 1950 through 1959). However, specifying the format-transform function prevents the use of an index for DOB field values:

ObjectScript

```
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE %ODBCOUT(DOB) %MATCHES '195*'"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

Arguments

scalar-expression

A scalar expression (most commonly a data column) whose values are being compared with *pattern*.

string

A quoted string representing the pattern of characters to match with each value in *scalar-expression*. The *pattern* string can contain literal characters, the question mark (?) and asterisk (*) wildcard characters, square brackets used to specify allowed values, and the backslash (\) used to specify that the character immediately following it is to be treated as a literal. The *pattern* can also be the empty string or NULL, though it does not match or return NULL items.

ESCAPE char

An optional argument. A string containing a single character. This *char* character can be used in *pattern* to specify that the character immediately following it is to be treated as a literal. If not specified, the default escape character is backslash (\).

Examples

The following example returns all last names that begin with "A":

SQL

```
SELECT Name FROM Sample.Person
WHERE Name %MATCHES 'A*'
```

The following example returns all first names that begin with "A":

SQL

```
SELECT Name FROM Sample.Person
WHERE Name %MATCHES '*,A*'
```

The following example returns all names that contain the letter "A" (in last name, first name, or middle initial):

SQL

```
SELECT Name FROM Sample.Person
WHERE Name %MATCHES '*A*'
```

The following example returns all names that *do not* contain the letters "A", "a", "E" or "e":

SQL

```
SELECT Name FROM Sample.Person
WHERE Name NOT %MATCHES '*[AaEe]*'
```

The following example returns all five-letter last names with first names that begin with "A" through "D":

SQL

```
SELECT Name FROM Sample.Person
WHERE Name %MATCHES '?????,[A-D]*'
```

See Also

- SELECT statement, HAVING clause, WHERE clause

- LIKE predicate

- %PATTERN predicate

- Overview of Predicates

%PATTERN (SQL)

Matches a value with a pattern string containing literals, wildcards, and character type codes.

Synopsis

```
scalar-expression %PATTERN pattern
```

Description

The **%PATTERN** predicate allows you to match a pattern of character type codes and literals to the data values supplied by *scalar-expression*. If *pattern* matches a complete scalar expression value, this value is returned. If *pattern* does not fully match any of the scalar expression values, **%PATTERN** returns the null string.

%PATERN can be used wherever a predicate condition can be specified, as described in Overview of Predicates.

%PATTERN uses the same pattern codes as the ObjectScript pattern match operator (the ? operator). A pattern consists of one or more pairs of a repetition count followed by a value. A repetition count can be an integer, a period (.) meaning "any number of characters", or a range specified by using a combination of a period with integers. A value can be either a character type code letter or a literal string (specified in quotes).

Note that a pattern often consists of multiple repetition/value pairs, because the pattern must exactly match the entire data value. For this reason, many patterns end with the ".E" pair, which means that the rest of the data value can consist of any number of characters of any type.

A few simple examples of pattern match pairs:

* 1L means one (and only one) lowercase letter.

* 1"L" means one literal character "L".

* 1"617" means one literal string "617".

* .U means any number of uppercase letters.

* .E means any number of printable characters of any type.

* .3A means any number up to three (three or less) letters (either uppercase or lowercase).

* 3.N means three or more numeric digits.

* 3.6N means three to six (inclusive) numeric digits.

Pattern matches are case-sensitive. Pattern matching is based on the EXACT value of *scalar-expression*, not its collation value. Therefore, a literal letter specified in a **%PATTERN** operation is always matched case-sensitive, even when the collation type of *scalar-expression* is not case-sensitive.

In Dynamic SQL the SQL query is specified as an ObjectScript string, delimited by double quotes. For this reason, double quotes within a *pattern* string must be doubled. Thus the pattern for a US dollar amount: `1"$"1.N1"."2N` would be specified in Dynamic SQL as `1""$""1.N1""."".""2N`.

For further details on pattern codes, refer to the Pattern Match Operator reference page.

%SelectMode

The **%PATTERN** predicate does not use the current %SelectMode setting. A *pattern* should be specified in Logical format, regardless of the %SelectMode setting. Attempting to specify a *pattern* in ODBC format or Display format commonly results in no data matches or unintended data matches.

You can use the %EXTERNAL or %ODBCOUT format-transform functions to transform the *scalar-expression* field that the predicate operates upon. This allows you to specify the *pattern* in Display format or ODBC format. However, using a format-transform function prevents the use of the index for the field, and can thus have a significant performance impact.

In the following Dynamic SQL example, the **%PATTERN** predicate specifies the date *pattern* in Logical format, not in %SelectMode=1 (ODBC) format. Rows with DOB Logical values beginning with 41 (dates from April 4 1953 ($HOROLOG 41000) through December 28 1955 ($HOROLOG 41999)) are selected:

ObjectScript

```
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE DOB %PATTERN '1""41""3N' "
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

The following Dynamic SQL example uses the %ODBCOUT format-transform function to transform the DOB field matched by the predicate. This allows you to specify the **%PATTERN** *pattern* in ODBC format. It selects rows with DOB field ODBC values beginning with 195 (dates within the range of years 1950 through 1959). However, specifying the format-transform function prevents the use of an index for DOB field values:

ObjectScript

```
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE %ODBCOUT(DOB) %PATTERN '1""195"".E' "
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

Arguments

scalar-expression

A scalar expression (most commonly a data column) whose values are being compared with *pattern*.

pattern

A quoted string representing the pattern of characters to match with each value in *scalar-expression*. The *pattern* string can contain literal characters enclosed in double quotes, letter codes that specify types of characters, and numbers and the period (.) character as wildcard characters.

Examples

The following example uses a **%PATTERN** operator in the **WHERE** clause to select Home_State values in which the first character is any uppercase letter and the second character is the letter "C":

SQL

```
SELECT Name,Home_State FROM Sample.Person
WHERE Home_State %PATTERN '1U1"C"'
```

This example selects records with a Home_State of North Carolina (NC) or South Carolina (SC).

The following example uses a **%PATTERN** operator in the **WHERE** clause to select Name values that start with an uppercase letter followed by a lowercase letter.

SQL

```
SELECT Name FROM Sample.Person
WHERE Name %PATTERN '1U1L.E'
```

The pattern here translates as: 1U (one uppercase letter), followed 1L (one lowercase letter), followed by .E (any number of characters of any type). Note that this pattern would exclude names such as "JONES", O'Reilly" and "deGastyne".

The following example uses a **%PATTERN** operator in a **HAVING** clause to select records for people whose first name starts with the letters "Jo", and to return the count of records searched and records returned.

SQL

```
SELECT Name,
       COUNT(Name) AS TotRecs,
       COUNT(Name %AFTERHAVING) AS JoRecs
FROM Sample.Person
HAVING Name %PATTERN '1U.L1","1"Jo".E'
```

In this case, the Name field values are formatted as Lastname,Firstname and may contain an optional middle name or initial. To reflect this name format, the pattern here translates as: 1U (one uppercase letter), followed .L (any number of lowercase letters), followed by 1"," (one literal comma character), followed by 1"Jo" (one literal string with the value "Jo"), followed by .E (any number of characters of any type).

See Also

- SELECT statement, HAVING clause, WHERE clause

- LIKE predicate

- %MATCHES predicate

- Overview of Predicates

SOME (SQL)

Matches a value with at least one matching value from a subquery.

Synopsis

```
scalar-expression comparison-operator SOME (subquery)
```

Description

The **SOME** keyword works in conjunction with a comparison operator to create a predicate (a quantified comparison condition) that is true if the value of a scalar expression matches one or more of the corresponding values retrieved by the subquery. The **SOME** predicate compares a single *scalar-expression* item with a single subquery **SELECT** item. A subquery with more than one select item generates an SQLCODE -10 error.

Note: The **SOME** and **ANY** keywords are synonyms.

SOME can be used wherever a predicate condition can be specified, as described in Overview of Predicates.

Arguments

scalar-expression

A scalar expression (most commonly a data column) whose values are being compared with the result set generated by *subquery*.

comparison-operator

One of the following comparison operators: = (equal to), <> or != (not equal to), < (less than), <= (less than or equal to), > (greater than), >= (greater than or equal to), [(contains), or] (follows).

subquery

A subquery, enclosed in parentheses, which returns a result set that is used for the comparison with *scalar-expression*.

Example

The following example selects those employees with salaries greater than $75,000 that live in any of the states west of the Mississippi River:

SQL

```
SELECT Name,Salary,Home_State FROM Sample.Employee
WHERE Salary > 75000
AND Home_State = SOME
 (SELECT State FROM Sample.USZipCode
  WHERE Longitude < -93)
ORDER BY Home_State
```

See Also

- SELECT statement, HAVING clause, WHERE clause

- ALL ANY

- Overview of Predicates

%STARTSWITH (SQL)

Matches a value with a substring specifying initial characters.

Synopsis

```
column %STARTSWITH substring
```

Description

- *column* **%STARTSWITH** *substring* selects data values from a column that begin with the characters specified in *substring*. If *substring* does not match any column values, **%STARTSWITH** returns the null string. **%STARTSWITH** performs this match on the logical, internal storage value of the column, regardless of the display mode set.

  You can use **%STARTSWITH** in any predicate condition of an InterSystems SQL query. For more details on predicate conditions, see Overview of Predicates.

  This statement selects all names that begin with the letter M.

  SQL

  ```
  SELECT Name FROM Sample.MyTest WHERE Name %STARTSWITH 'M'
  ```

  For other ways of matching a value, see Other Equivalence Comparisons.

  Examples:

  – Select Column Data Based on Initial Characters

  – Select Column Data Using Logical Operators

Arguments

column

A data column in a table whose values are being compared with *substring*. This argument can also be a scalar expression that evaluates to a column table, such as `%EXTERNAL(column)` or `%SQLUPPER(column)`.

substring

The first character or characters to match with values in *column*. This argument must be an expression that resolves to a string or numeric value.

Examples

Select Column Data Based on Initial Characters

The **%STARTSWITH** predicate can process a variety of string and numeric types.

Letters

This statement returns one row for each distinct Home_State name that begins with the letter M.

SQL

```
SELECT DISTINCT Home_State FROM Sample.Person
WHERE Home_State %STARTSWITH 'M'
ORDER BY Home_State
```

Under the default collation settings, %STARTSWITH matches are not case-sensitive, so this statement matches names beginning with either "M" or "m". For more details on controlling the case-sensitivity of matches, see Manage Case-Sensitivity of Selections Based on Collation Type.

Numbers

This statement uses a **HAVING** clause to select rows for people whose age starts with a 2. The result set displays the average for all ages and the average for the ages selected by the **HAVING** clause. It orders the results by age.

SQL

```
SELECT Name,
       Age,
       AVG(Age) AS AvgAge,
       AVG(Age %AFTERHAVING) AS Avg20
FROM Sample.Person
HAVING Age %STARTSWITH 2
ORDER BY Age
```

Dates

This statement performs a **%STARTSWITH** comparison with the internal date format value for the DOB (date of birth) field. In this case, it select all dates from 11/5/1988 (**$H**=54000) through 08/1/1991 (**$H**=54999):

SQL

```
SELECT Name,DOB
FROM Sample.Person
WHERE DOB %STARTSWITH 54
ORDER BY DOB
```

Lists

If *column* contains a list collection, **%STARTSWITH** can use the **%EXTERNAL** format transformation function to compare the list values to *substring*. For example, this statement matches on rows in which the FavoriteColors list column begins with 'Bl':

SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE %EXTERNAL(FavoriteColors) %STARTSWITH 'Bl'
```

For list collections, when **%EXTERNAL** converts a list to DISPLAY format, the displayed list items appear to be separated by a blank space. This "space" is actually the two non-display characters CHAR(13) and CHAR(10). To use **%STARTSWITH** with more than one element in the list, you must specify these characters:

SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE %EXTERNAL(FavoriteColors) %STARTSWITH 'Orange'||CHAR(13)||CHAR(10)||'B'
```

If *column* contains data of type %Libary.List, you do not need to use the **%EXTERNAL**, because %Libary.List data is stored in the LOGICAL format and does not have a separate DISPLAY format. To match list data, you can use $LIST functions. For example:

SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FavoriteColors %STARTSWITH $LISTFROMSTRING('Yellow,Orange')
```

Note: Because InterSystems SQL stores lists as concatenated strings, you cannot use %STARTSWITH to match on elements in the middle of a list. %STARTSWITH matches only from the start of the list. This applies to both list collections and %Libary.List data.

Leading and Trailing Blanks

In most cases, **%STARTSWITH** treats leading blanks the same as any other character. For example, `%STARTSWITH '
B'` selects column values with exactly one leading blank followed by the letter B. However, a *substring* containing only blanks selects non-null values, not leading blanks.

The **%STARTSWITH** behavior with trailing blanks depends on the data type and collation type:

- **%STARTSWITH** ignores trailing blanks in a string *substring* with SQLUPPER collation.

- **%STARTSWITH** does not ignore trailing blanks in a numeric, date, or list *substring*.

In this statement, **%STARTSWITH** restricts the result set to names that begin with 'M'. Because Name is an SQLUPPER string data type, the trailing blanks in the *substring* are ignored.

SQL

```
SELECT Name FROM Sample.Person
WHERE Name %STARTSWITH 'M      '
```

In this statement, **%STARTSWITH** eliminates all rows from the result set because the trailing blanks in the *substring* are not ignored for a numeric value:

SQL

```
SELECT Name,Age FROM Sample.Person
WHERE Age %STARTSWITH '6      '
```

In this statement, **%STARTSWITH** eliminates all rows from the result set because the trailing blank in the *substring* is not ignored for a list value:

SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE %EXTERNAL(FavoriteColors) %STARTSWITH 'Blue '
```

However, in this statement, the result set consists of those list values that start with Blue followed by a list delimiter, which is displayed as a blank space. In other words, this statement matches on lists beginning with 'Blue' that contain more than one item:

SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE %EXTERNAL(FavoriteColors) %STARTSWITH 'Blue'||CHAR(13)||CHAR(10)
```

Select Column Data Using Logical Operators

The **%STARTSWITH** function supports the logical operators NOT, AND, and OR.

This statement selects all names that do not begin with the letter M.

SQL

```
SELECT Name FROM Sample.MyTest WHERE NOT Name %STARTSWITH 'M'
```

This statement selects all names that begin with M or N.

SQL

```
SELECT Name FROM Sample.MyTest WHERE Name %STARTSWITH 'M' OR Name %STARTSWITH 'N'
```

This statement selects all names in which the first names begin with M and the last names begin with N.

SQL

```
SELECT FirstName,LastName FROM Sample.MyTest WHERE FirstName %STARTSWITH 'M' AND LastName %STARTSWITH
 'N'
```

Filter Out Null Values

In the **%STARTSWITH** function, if *column* evaluates to a non-null data value and *substring* is an empty value, then **%STARTSWITH** returns the non-null *column* data. You can use this behavior to filter out non-null values. For example, this statement restricts the result set to non-null values in the FavoriteColors column.

SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FavoriteColors %STARTSWITH NULL
```

An empty *substring* can be any of these values:

- NULL

- CHAR(0)

- the empty string ('')

- a string consisting of only blank spaces (' ')

- CHAR(32), which is the space character

- CHAR(9), which is the tab character

These statements return the same results as the previous statement.

SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FavoriteColors %STARTSWITH ''
```

SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FavoriteColors %STARTSWITH '    '
```

SQL

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FavoriteColors %STARTSWITH CHAR(9)
```

If *column* evaluates to null and *substring* is an empty value, **%STARTSWITH** does not return data from *column*.

To return *column* values that consist of only whitespace characters, you must use %EXACT collation. Note that the **%EXTERNAL** collation type is not used for *column* when filtering nulls from a list field.

%STARTSWITH NULL and empty string behavior differs with a compound *substring*, because of the definitions of NULL and empty string. When you concatenate a value with NULL, the result is NULL. When you concatenate a value with the empty string, the result is the value. This is shown in the following examples:

SQL

```
SELECT Name,FavoriteColors
FROM Sample.Person
WHERE %EXTERNAL(FavoriteColors) %STARTSWITH 'B'||NULL
/* Selects all non-null rows */
```

SQL

```
SELECT Name,FavoriteColors
FROM Sample.Person
WHERE %EXTERNAL(FavoriteColors) %STARTSWITH 'B'||''
/* Selects all values that begin with B */
```

Manage Case-Sensitivity of Selections Based on Collation Type

%STARTSWITH uses the same collation as the field it is matched against. Since string data type fields are defined with the SQLUPPER collation, which is not case-sensitive, the default **%STARTSWITH** selections are also not case-sensitive. For example, this statement matches home states beginning with either "M" or "m".

SQL

```
SELECT DISTINCT Home_State FROM Sample.Person
WHERE Home_State %STARTSWITH 'M'
ORDER BY Home_State
```

If you assign a different collation type to the column in the WHERE clause, this collation type is matched to the literal value of the **%STARTSWITH** *substring*. For example, if you specify the search column, Home_State, to use EXACT (case-sensitive) collation, then **%STARTSWITH** matches only on home states beginning with "M".

SQL

```
SELECT DISTINCT Home_State FROM Sample.Person
WHERE %EXACT(Home_State) %STARTSWITH 'M'
ORDER BY Home_State
```

Some collation functions prepend a space character to a field value. This can cause **%STARTSWITH** to match no values, unless you apply an equivalent collation function to the substring.

For example, suppose a table contains a column, `ExactName`, that uses EXACT collation. If you apply SQLUPPER collation to `ExactName` within the *column* argument of **%STARTSWITH**, then **%STARTSWITH** searches a column whose values all start with a space character. Therefore, a comparison such as this one returns no rows:

SQL

```
SELECT ExactName FROM Sample.MyTest WHERE %SQLUPPER(ExactName) %STARTSWITH 'Ra'
```

To resolve this issue, you must prepend a space character to the *substring*, such as by applying the same collation function to the substring. This example applies a case-insensitive match to an EXACT column:

SQL

```
SELECT ExactName FROM Sample.MyTest WHERE %SQLUPPER(ExactName) %STARTSWITH %SQLUPPER('Ra')
```

For details on changing the collation defaults or using case transformation functions, see Collation.

More About

Range of Subscripts

When *column* is retrieved from a subscript, **%STARTSWITH** can be used as an index-limiting range condition, narrowing the range of *column* subscript values that needs to be traversed. The logic is to start the subscript range with the given *substring* prefix value, and stop as soon as the subscript value no longer starts with *substring*.

Other Equivalence Comparisons

%STARTSWITH performs an equivalence comparison on the initial characters of a string. You can perform other types of equivalence comparisons by using string comparison operators. These include the following:

- An equivalence comparison on the entire string, using the equal sign operator:

SQL

```
SELECT Name,Home_State FROM Sample.Person
WHERE Home_State = 'VT'
```

This example selects any record that contains the Home_State field value "VT". Because Home_State is defined as SQLUPPER, this string comparison is not case-sensitive.

You can also perform a non-equivalence comparison on the entire string, using the not equal operator (<>).

- An equivalence comparison of a substring to a value, using the Contains operator:

SQL

```
SELECT Name FROM Sample.Person
WHERE Name [ 'y'
```

This example selects all Name records that contain the lowercase letter "y". By default, a Contains operator comparison is case-sensitive, even when the field is defined as not case-sensitive.

- A context-aware equivalence comparison using InterSystems SQL Search. One use of SQL Search is to determine if a value contains a specified word or phrase. SQL Search is not case-sensitive.

- An equivalence comparison on the entire string to multiple values, using the IN keyword operator. For example, this statement selects any record that contains any of the specified Home_State field values.

SQL

```
SELECT Name,Home_State FROM Sample.Person
WHERE Home_State IN ('VT','MA','NH','ME')
ORDER BY Home_State
```

- An equivalence comparison on the entire string to a value pattern, using the **%PATTERN** keyword operator:

SQL

```
SELECT Name,Home_State FROM Sample.Person
WHERE Home_State %PATTERN '1U1"C"'
ORDER BY Home_State
```

This example selects any record that contains a Home_State field value that matches the pattern of 1U (one uppercase letter) followed by 1"C" (one literal letter "C"). The Home_State abbreviations "NC" or "SC" fulfill this pattern.

- An equivalence comparison of a substring with one or more wildcards to a value, using the LIKE keyword operator:

SQL

```
SELECT Name FROM Sample.Person
WHERE Name LIKE '_a%'
```

This example selects all Name records that contain the letter "a" as the second letter. This string comparison uses the Name collation type to determine whether the comparison is case-sensitive or not.

For further details on these and other comparison conditional predicates, refer to the WHERE clause.

%SelectMode Setting

The **%STARTSWITH** predicate cannot use the current %SelectMode setting. A *substring* must be specified in Logical format, regardless of the %SelectMode setting. Specifying predicate value(s) in ODBC or Display format commonly results in no data matches or unintended data matches. This applies mainly to dates, times, and InterSystems IRIS format lists (%List).

In the following Dynamic SQL example, the **%STARTSWITH** predicate must specify the date *substring* in Logical format, not in %SelectMode=1 (ODBC) format. Rows with DOB Logical values beginning with 41 (dates from April 4 1953 ($HOROLOG 41000) through December 28 1955 ($HOROLOG 41999)) are selected:

ObjectScript

```
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE DOB %STARTSWITH '41%'"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

National Collation of Ambiguous Characters

In some national languages two characters or character combinations are considered first-pass collation equivalent. Commonly this is a character with or without an accent mark, such as in the Czech2 locale, in which CHAR(65) and CHAR(193) both collate as "A". **%STARTSWITH** recognizes these characters as equivalent.

The following example shows the first-pass collation for Czech2 CHAR(65) (A) and CHAR(193) (Á):

```
M
MA
MÁ
MAC
MÁC
MACX
MÁCX
MAD
MÁD
MB
```

When the query compiles, the national collation used at run time is unknown. Therefore, write **%STARTSWITH** subscript traversal code to satisfy the possible runtime scenarios.

See Also

- SELECT statement, HAVING clause, WHERE clause

- Overview of Predicates

- Collation

SQL Aggregate Functions

Overview of Aggregate Functions

Functions that evaluate all of the values of a column and return a single aggregate value.

Supported Aggregate Functions

An aggregate function performs a task in relation to one or more values from a single column and returns a single value. The supported functions are:

- SUM — returns the sum of the values of a specified column.

- AVG — returns the average of the values of the specified column.

- COUNT — returns the number of rows in a table, or the number of non-null values in a specified column.

- MAX — returns the maximum value used within a specified column.

- MIN — returns the minimum value used within a specified column.

- VARIANCE, VAR_SAMP, VAR_POP — returns the statistical variance of the values of a specified column.

- STDDEV, STDDEV_SAMP, STDDEV_POP — returns the statistical standard deviation of the values of a specified column.

- LIST — returns all of the values used within a specified column as a comma-separated list.

- %DLIST — returns all of the values used within a specified column as elements in an InterSystems IRIS list structure.

- XMLAGG — returns all of the values used within a specified column as a concatenated string.

- JSON_ARRAYAGG — returns all of the values used within a specified column as a JSON format array.

You can define additional user-defined aggregate functions (UDAFs) using the CREATE AGGREGATE command.

Aggregate functions ignore fields that are NULL. For example, **LIST** and **%DLIST** do not include elements for rows in which the specified field is NULL. **COUNT** only counts non-null values of the specified field.

Aggregate functions (with the exception of **COUNT**) cannot be applied to a stream field. Attempting to do so generates an SQLCODE -37 error. You can use **COUNT** to count stream field values, with some restrictions.

Note: Aggregate functions are similar to window functions. However, aggregate functions take the values of a column from a group of rows and return the result as a single value. Window functions take the values of a column from a group of rows and return a value for each row. An aggregate function can be specified in a window function. A window function cannot be specified in an aggregate function. AVG(), MAX(), MIN(), and SUM() can be used as either aggregate functions or window functions.

Using Aggregate Functions

An aggregate function can be used in:

- SELECT list, either as a listed *selectItem* or in a subquery *selectItem*.

- HAVING clause. However, a HAVING clause must explicitly specify the aggregate function; it cannot specify an aggregate using the corresponding *selectItem* column alias or *selectItem* sequence number.

- DISTINCT BY clause. However, specifying an aggregate function by itself is not meaningful and always returns a single row. More meaningful is to specify an aggregate function as part of an expression, such as `DISTINCT BY(MAX(Age)-Age)`.

An aggregate function *cannot* be used directly in:

- an ORDER BY clause. Attempting to do so generates an SQLCODE -73 error. However, you can use an aggregate function in an ORDER BY clause by specifying the corresponding column alias or *selectItem* sequence number.

- a WHERE clause. Attempting to do so generates an SQLCODE -19 error.

- a GROUP BY clause. Attempting to do so generates an SQLCODE -19 error.

- a TOP clause. Attempting to do so generates an SQLCODE -1 error.

- a JOIN. Attempting to specify an aggregate in an ON clause generates an SQLCODE -19 error. Attempting to specify an aggregate in a USING clause generates an SQLCODE -1 error.

However, you can supply an aggregate function value to these clauses (with the exception of the TOP clause) by using a subquery supplying a column alias. For example, to use a WHERE clause to select Age values that are less than the average Age value, you can place the **AVG** aggregate function in a subquery:

SQL

```
SELECT Name,Age,AvgAge
FROM (SELECT Name,Age,AVG(Age) AS AvgAge FROM Sample.Person)
WHERE Age < AvgAge
ORDER BY Age
```

Combining Aggregates and Fields

InterSystems SQL allows you to specify an aggregate function with other SELECT items in a query. An aggregate such as COUNT(*) does not need to be in a separate query.

SQL

```
SELECT TOP 5 COUNT(*),Name,AVG(Age)
FROM Sample.Person
ORDER BY Name
```

When you specify an aggregate function and specify no field select items in the select list, InterSystems SQL returns one row. A TOP clause is ignored, unless it is TOP 0 (return no rows):

SQL

```
SELECT TOP 7 AVG(Age),LIST(Age)
FROM Sample.Person
WHERE Age > 75
```

When you specify an aggregate function and specify one or more field select items in the select list, InterSystems SQL returns as many rows as required for the field item:

SQL

```
SELECT DISTINCT Age,AVG(Age),LIST(Age)
FROM Sample.Person
WHERE Age > 75
```

Column Names and Aliases

By default, the column name assigned to the results of an aggregate function is Aggregate_*n*, where the *n* number suffix is the column order number, as specified in the **SELECT** list. Thus, the following example creates column names Aggregate_2 and Aggregate_5:

SQL

```
SELECT TOP 5 Home_State,COUNT(*),Name,Age,AVG(Age)
FROM Sample.Person
ORDER BY Name
```

To specify another column name (a column alias), use the AS keyword:

SQL

```
SELECT COUNT(*) AS PersonCount
FROM Sample.Person,Sample.Employee
```

You can use a column alias to specify an aggregate field in an ORDER BY clause. The following example lists people in the order that their ages diverge from the average age:

SQL

```
SELECT Name,Age,
    AVG(Age) AS AvgAge,
    ABS(Age - AVG(Age)) AS RelAge
FROM Sample.Person
ORDER BY RelAge
```

For further details on column aliases, refer to the **SELECT** statement.

With ORDER BY

The **LIST**, **%DLIST**, **XMLAGG**, and **JSON_ARRAYAGG** functions combine the values of a table column from multiple rows into a single aggregate value. Because an **ORDER BY** clause is applied to the query result set after all aggregate fields are evaluated, **ORDER BY** cannot directly affect the sequence of values within these aggregates. Under certain circumstances, the results of these aggregates may appear in sequential order, but this ordering should not be relied upon. The values listed within a given aggregate result value cannot be explicitly ordered.

DISTINCT Keyword Clause

All aggregate functions support the optional DISTINCT keyword clause. This keyword limits the aggregate operation to only distinct (unique) field values. When using default field collation (%SQLUPPER), field values that differ only in lettercase are not considered distinct values. If DISTINCT is not specified, the default is to perform the aggregate operation on all non-NULL values, including duplicate values. The **MIN** and **MAX** aggregate functions support the DISTINCT keyword, although it perform no operation.

Aggregate functions support the full DISTINCT keyword clause syntax, including the optional `BY(item-list)` subclause. Refer to the DISTINCT clause for details.

The aggregate function DISTINCT field1 clause ignores field1 values that are NULL. This differs from the DISTINCT clause of the SELECT statement: a SELECT DISTINCT clause returns one row for the distinct NULL, just as it returns one row for each distinct field value. However, an aggregate function DISTINCT BY(field2) field1 does not ignore the distinct NULL for field2. For example, if FavoriteColors has 50 distinct values and multiple NULLs, the number of DISTINCT rows returned is 51, the `COUNT(DISTINCT FavoriteColors)` is 50, and the `COUNT(DISTINCT BY(FavoriteColors) %ID)` is 51:

SQL

```
SELECT DISTINCT FavoriteColors,
      COUNT(DISTINCT FavoriteColors),
      COUNT(DISTINCT BY(FavoriteColors) %ID)
 FROM Sample.Person
```

With DISTINCT and GROUP BY

A **SELECT** DISTINCT with a *selectItem* aggregate function and a **GROUP BY** clause returns the same results as if the DISTINCT keyword were not present. To achieve the desired results, put the aggregate function in a subquery.

For example, you wish to return the number of distinct counts of persons in states (there are states with 4 people, there are states with 6 people, etc.). You would expect to achieve this result as follows:

SQL

```
SELECT DISTINCT COUNT(*) AS PersonCounts
FROM Sample.Person
GROUP BY Home_State
```

Instead, you get a person count for each state, the same as if the DISTINCT keyword were not present:

SQL

```
SELECT COUNT(*) AS PersonCounts
FROM Sample.Person
GROUP BY Home_State
```

To achieve your intended result, you need to use a subquery, as follows:

SQL

```
SELECT DISTINCT *
FROM (SELECT COUNT(*) AS PersonCounts FROM Sample.Person
     GROUP BY Home_State)
```

Row Counts

When a query returns aggregate values, the %ROWCOUNT value depends on the query:

- Aggregate functions only: calculates aggregate values and returns %ROWCOUNT 1. If an aggregates-only query selects no rows, it still returns %ROWCOUNT 1: **COUNT**=0, other aggregate functions return NULL.

- Aggregate functions only with GROUP BY: returns aggregate values for each group selected by the GROUP BY clause. %ROWCOUNT is the number of groups selected. If the query selects no rows, the GROUP BY selects no groups, and the query returns %ROWCOUNT 0.

- Aggregate functions only with DISTINCT: calculates aggregate values and returns %ROWCOUNT 1. If the query selects no rows, the DISTINCT selects no distinct values, and the query returns %ROWCOUNT 0.

- Aggregate functions only with TOP clause: For any non-zero TOP value, calculates aggregate values and returns %ROWCOUNT 1. For TOP=0, returns %ROWCOUNT 0, aggregates are not calculated.

- Aggregates with fields: If the query returns field values as well as aggregate functions, the number of rows returned is the number of rows selected. If the query selects no rows, it returns %ROWCOUNT 0 and aggregates are not calculated.

These results are not affected the presence in the *selectItem* of subqueries or expressions.

Aggregates, Transactions, and Locking

Including an aggregate function in a query causes the query to return the current state of the data to all result set fields, including uncommitted changes to the data. Thus, an ISOLATION LEVEL READ COMMITTED setting is ignored for a query containing an aggregate function. The current state of uncommitted data is as follows:

- INSERT and UPDATE: the aggregate calculation *does* include the modified values, even though these modifications are not yet committed and may be rolled back.

- DELETE and TRUNCATE TABLE: the aggregate calculation *does not* include deleted rows, even though these deletions are not yet committed and may be rolled back.

Because aggregate functions usually involve data from a large number of rows, it is not acceptable to issue a transaction lock on all of the rows involved in an aggregate calculation. It is therefore possible that another user may be performing a transaction that modifies the data while an aggregate calculation is in process.

Aggregates and Sharded Tables

Support for aggregate functions is limited for sharded tables. For example, the aggregate function DISTINCT, %FOREACH, and %AFTERHAVING clauses are not supported for sharded tables. See Querying the Sharded Cluster.

See Also

- AVG, COUNT, %DLIST, JSON_ARRAYAGG, LIST, MAX, MIN, STDDEV, STDDEV_SAMP, STDDEV_POP, SUM, VARIANCE, VAR_SAMP, VAR_POP, XMLAGG aggregate functions

- CREATE AGGREGATE command

- SELECT command

- Overview of Window Functions

AVG (SQL)

An aggregate function that returns the average of the values of the specified column.

Synopsis

```
AVG([ALL | DISTINCT [BY(col-list)]] expression [%FOREACH(col-list)] [%AFTERHAVING])
```

Description

The **AVG** aggregate function returns the average of the values of *expression*. Commonly, *expression* is the name of a field, (or an expression containing one or more field names) in the multiple rows returned by a query.

AVG can be used in a SELECT query or subquery that references either a table or a view. **AVG** can appear in a **SELECT** list or **HAVING** clause alongside ordinary field values.

AVG cannot be used in a **WHERE** clause. **AVG** cannot be used in the **ON** clause of a **JOIN**, unless the **SELECT** is a subquery.

AVG, like all aggregate functions, can take an optional DISTINCT clause. AVG(DISTINCT col1) averages only those col1 field values that are distinct (unique). AVG(DISTINCT BY(col2) col1) averages only those col1 field values in records where the col2 values are distinct (unique). Note however that the distinct col2 values may include a single NULL as a distinct value.

Data Values

For non-DOUBLE *expression* values, **AVG** returns a double-precision floating point number. The precision of the value returned by **AVG** is 18. The scale of the returned value depends upon the precision and scale of *expression*: the scale of the value returned by **AVG** is equal to 18 minus the *expression* precision, plus the *expression* scale.

For DOUBLE *expression* values, the scale is 0.

AVG is normally applied to a field or expression that has a numeric value, such as a number field or a date field. By default, aggregate functions use Logical (internal) data values, rather than Display values. Because no type checking is performed, it is possible (though rarely meaningful) to invoke it for nonnumeric fields; **AVG** evaluates nonnumeric values, including the empty string ("), as zero (0). If *expression* is data type VARCHAR, the return value is data type DOUBLE.

NULL values in data fields are ignored when deriving an **AVG** aggregate function value. If no rows are returned by the query, or the data field value for all rows returned is NULL, **AVG** returns NULL.

Averaging a Single Value

If all of the *expression* values supplied to **AVG** are the same, the resulting average depends on the number of accessed rows in the table (the divisor). For example, if all of the rows in the table have the same value for a specific column, the average value of that column is a calculated value, which may differ slightly from the value in the individual columns.

The following example shows how a slight inequality can result from the calculation of an average. The first query does not reference table rows, so **AVG** calculates by dividing by 1. The second query references table rows, so **AVG** calculates by dividing by the number of rows in the table.

SQL

```
SELECT
  {fn PI} AS Pi,
  AVG({fn PI}) AS AvgPiDividedBy1
FROM Sample.Person
```

SQL

```
SELECT
  Name,
  {fn PI} AS Pi,
  AVG({fn PI}) AS AvgPiDividedByNumRows
FROM Sample.Person
```

Optimization

SQL optimization of an **AVG** calculation can use a bitslice index, if this index is defined for the field.

Changes Made During the Current Transaction

Like all aggregate functions, **AVG** always returns the current state of the data, including uncommitted changes, regardless of the current transaction's isolation level. For further details, refer to SET TRANSACTION and START TRANSACTION.

Arguments

ALL

An optional argument specifying that **AVG** return the average of all values for *expression*. This is the default if no keyword is specified.

DISTINCT

An optional DISTINCT clause that specifies that **AVG** calculate the average on only the unique instances of a value. DISTINCT can specify a BY(col-list) subclause, where *col-list* can be a single field or a comma-separated list of fields.

expression

Any valid expression. Usually the name of a column that contains the data values to be averaged.

%FOREACH(col-list)

An optional column name or a comma-separated list of column names. See SELECT for further information on %FOREACH.

%AFTERHAVING

An optional argument that applies the condition found in the HAVING clause.

AVG returns either the NUMERIC or DOUBLE data type. If *expression* is data type DOUBLE, **AVG** returns DOUBLE; otherwise, it returns NUMERIC.

Examples

The following query lists the average salary for all employees in the Sample.Employee database. Because all rows returned by the query would have identical values for this average, this query only returns a single row, consisting of the average salary. For display purposes, this query concatenates a dollar sign to the value (using the || operator), and uses the AS clause to label the column:

SQL

```
SELECT '$' || AVG(Salary) AS AverageSalary
    FROM Sample.Employee
```

The following query lists each state with the average salary for the employees in that state:

SQL

```
SELECT Home_State,'$' || AVG(Salary) AS AverageSalary
    FROM Sample.Employee
GROUP BY Home_State
```

The following query lists the name and salary for those employees whose salary is greater than the average salary. It also lists the average salary for all employees; this value is the same for all rows returned by the query:

SQL

```
SELECT Name,Salary,
       '$' || AVG(Salary) AS AverageAllSalary
FROM Sample.Employee
HAVING Salary>AVG(Salary)
ORDER BY Salary
```

The following query lists the name and salary for those employees whose salary is greater than the average salary. It also lists the average salary for those employees with above-average salaries; this value is the same for all rows returned by the query:

SQL

```
SELECT Name,Salary,
       '$' || AVG(Salary %AFTERHAVING) AS AverageHighSalary
FROM Sample.Employee
HAVING Salary>AVG(Salary)
ORDER BY Salary
```

The following query lists those states containing more than three employees with the average salary of that state's employees, and the average salary of that state's employees earning more than $20,000:

SQL

```
SELECT Home_State,
       '$' || AVG(Salary) AS AvgStateSalary,
       '$' || AVG(Salary %AFTERHAVING) AS AvgLargerSalaries
FROM Sample.Employee
GROUP BY Home_State
HAVING COUNT(*) > 3 AND Salary > 20000
ORDER BY Home_State
```

The following query uses several forms of the DISTINCT clause. The `AVG(DISTINCT BY col-list` examples may include an additional Age value in the average, because the BY clause can include a single NULL as a distinct value, if Home_City contains one or more NULLs:

SQL

```
SELECT AVG(Age) AS AveAge,AVG(ALL Age) AS Synonym,
       AVG(DISTINCT Age) AS AveDistAge,
       AVG(DISTINCT BY(Home_City) Age) AS AvgAgeDistCity,
       AVG(DISTINCT BY(Home_City,Home_State) Age) AS AvgAgeDistCityState
    FROM Sample.Person
```

The following query uses both the %FOREACH and the %AFTERHAVING keywords. It returns a row for those states containing people whose names start with "A", "M", or "W" (HAVING clause and GROUP BY clause). Each state row contains the following values:

- `LIST(Age %FOREACH(Home_State))`: a list of the ages of all of the people in the state.

- `AVG(Age %FOREACH(Home_State))`: the average age of all of the people in the state.

- `AVG(Age %AFTERHAVING)`: the average age of all of the people in the database that meet the HAVING clause criteria. (This number is the same for all rows.)

- `LIST(Age %FOREACH(Home_State) %AFTERHAVING)`: a list of the ages of all of the people in the state that meet the HAVING clause criteria.

- `AVG(Age %FOREACH(Home_State) %AFTERHAVING)`: the average age of all of the people in the state that meet the HAVING clause criteria.

SQL

```
SELECT Home_State,
       LIST(Age %FOREACH(Home_State)) AS StateAgeList,
       AVG(Age %FOREACH(Home_State)) AS StateAgeAvg,
       AVG(Age %AFTERHAVING ) AS AgeAvgHaving,
       LIST(Age %FOREACH(Home_State)%AFTERHAVING ) AS StateAgeListHaving,
       AVG(Age %FOREACH(Home_State)%AFTERHAVING ) AS StateAgeAvgHaving
FROM Sample.Person
GROUP BY Home_State
HAVING Name LIKE 'A%' OR Name LIKE 'M%' OR Name LIKE 'W%'
ORDER BY Home_State
```

See Also

- Aggregate Functions overview

- COUNT aggregate function

- SUM aggregate function

COUNT (SQL)

An aggregate function that returns the number of rows in a table or a specified column.

Synopsis

```
COUNT(*)
COUNT(expression)

COUNT(DISTINCT expression)
COUNT(DISTINCT BY(column) expression)
COUNT(ALL expression)

COUNT( ... expression %FOREACH(column))
COUNT( ... expression ... %AFTERHAVING)
```

Description

COUNT is an aggregate function that returns a count of the number of rows in a table or column. **COUNT** returns the BIGINT data type. If the count includes no rows, COUNT returns 0 or NULL, depending on the query. For more details, see No Rows Returned in Count.

Use **COUNT** in a SELECT query to count rows from the table referenced in the query and return the count in the result set. You can also use **COUNT** in a subquery that references either a table or view and in the **HAVING** clause. You cannot use **COUNT** in a **WHERE** clause. Unless **SELECT** is a subquery, you also cannot use **COUNT** in the **ON** clause of a **JOIN**.

- **COUNT(*)** returns the number of rows in the table or view. **COUNT(*)** counts all rows, including ones that contain duplicate column values or NULL values.

  This query returns the total number of rows in Sample.Person.

  SQL

  ```
  SELECT COUNT(*) FROM Sample.Person
  ```

  Example: Count Table Rows and Column Values

- **COUNT(*expression*)** returns the number of values in *expression*, which is a table column name or an expression that evaluates to a column of data. **COUNT(*expression*)** does not count NULL values.

  This query returns the number of non-NULL values in the Name column of Sample.Person.

  SQL

  ```
  SELECT COUNT(Name) AS TotalNames FROM Sample.Person
  ```

  Examples:

  - Count Table Rows and Column Values

  - Count Stream Column Values

  - Count Non-NULL Values in Combination of Columns

- **COUNT(DISTINCT *expression*)** uses a DISTINCT clause to return the count of the distinct (unique) values in the *expression* column. You cannot use DISTINCT with stream columns. What is counted as a distinct value depends on the column's collation. For example, with the default column collation of %SQLUPPER, values that differ in letter case are not counted as distinct values. To count every letter-case variant as a distinct value, use COUNT(DISTINCT(%EXACT(*expression*))). NULL values are not included in COUNT DISTINCT counts.

This statement returns the count of unique ages in `Sample.Person`.

SQL

```
SELECT COUNT(DISTINCT Age) FROM Sample.Person
```

Example: Count Distinct Column Values

- **COUNT(DISTINCT BY(*column*) *expression*)** filters out rows that contain duplicates in the column list specified by *column*, and then returns a count of values in the *expression* column. If *column* contains a NULL value, the NULL is counted as a distinct value.

This statement returns a count of `FavoriteColors` values for people with distinct (unique) names.

SQL

```
SELECT COUNT(DISTINCT BY(Name) FavoriteColors) FROM Sample.Person
```

Example: Count Distinct Column Values

- **COUNT(ALL *expression*)** returns the count of all values for *expression*. The ALL keyword counts all non-NULL values, including all duplicates. ALL is the default behavior if no keyword is specified.

- **COUNT(... *expression* %FOREACH(*column*))** groups the values in the *expression* column by the distinct values contained in the *column* list and returns the count of each group. %FOREACH and GROUP BY are similar. While **GROUP BY** operates on an entire query, `%FOREACH` allows selection of aggregates on sub-populations without restricting the entire query population.

This query returns a row for each person specified in `Sample.Person`, where each row contains that person's name, home state, and a count of the names of people who also live in that state.

SQL

```
SELECT
  Name,
  Home_State,
  COUNT(Name %FOREACH(Home_State)) AS PersonCountInState
FROM Sample.Person
```

Example: Count Grouped Values

- **COUNT(... *expression* ... %AFTERHAVING)** counts the rows in *expression* only after applying the condition found in the HAVING clause. If you omit **%AFTERHAVING**, the query does not account for the **HAVING** condition in its count.

This query returns the count of names grouped by state and the count of names that start with "M" grouped by state.

SQL

```
SELECT
  Home_State,
  COUNT(Name) AS NameCount,
  COUNT(Name %AFTERHAVING) AS MNameCount
FROM Sample.Person
GROUP BY Home_State
HAVING Name LIKE 'M%'
```

Example: Count Grouped Values

Arguments

expression

A valid expression that contains the data values to be counted. *expression* can be the name of a column or an expression that evaluates to a column of data. You cannot specify *expression* as a subquery.

column

A column name or comma-separated list of column names.

- In the **COUNT(*expression* %FOREACH(*column*))** syntax, *column* specifies the columns used to group the data before **COUNT** counts the values in the *expression* column. *column* cannot include a stream column.

- In the **COUNT(DISTINCT BY(*column*) *expression*)** syntax, *column* specifies the columns whose distinct values are used to filter out duplicate rows before **COUNT** counts the values in the *expression* column.

Examples

Count Table Rows and Column Values

This query returns the total number of rows in the `Sample.Person` table. The count includes rows containing NULL values in one or more columns.

SQL

```
SELECT COUNT(*) AS TotalPersons
FROM Sample.Person
```

This query returns the count of names, spouses, and favorite colors in `Sample.Person`. **COUNT** does not include NULL values in column counts. Therefore, the number of return values for each column might differ or be less than the total number of rows returned by **COUNT(*)**.

SQL

```
SELECT
  COUNT(Name) AS People,
  COUNT(Spouse) AS PeopleWithSpouses,
  COUNT(FavoriteColors) AS PeopleWithColorPref
FROM Sample.Person
```

Count Distinct Column Values

This query uses **COUNT DISTINCT** to return the count of distinct `FavoriteColors` values in `Sample.Person`. The `FavoriteColors` column contains several data values and multiple NULL. This query also uses the **SELECT DISTINCT** clause to return one row for each distinct `FavoriteColors` value. The row count is one larger than the `COUNT(DISTINCT FavoriteColors)` count. DISTINCT returns a row for a single NULL as a distinct value, but COUNT DISTINCT does not count NULL. The `COUNT(DISTINCT BY(FavoriteColors) %ID)` value is the same as the row count, because the BY clause does count a single NULL as a distinct value.

SQL

```
SELECT
  DISTINCT FavoriteColors,
  COUNT(DISTINCT FavoriteColors) AS DistColors,
  COUNT(DISTINCT BY(FavoriteColors) %ID) AS DistColorPeople
FROM Sample.Person
```

Count Grouped Values

The queries in this example use GROUP BY to group repeated values in a column, returning one row per unique value. The queries then use **COUNT** to return a per-group count of values from a different column.

This query returns one row per distinct `FavoriteColors` value. Assuming that `FavoriteColors` is not required, the query returns a row for NULL values (if any). Associated with each row are two counts:

- The number of rows that have that `FavoriteColors` option. Rows with NULL values are not counted.

- The number of names associated with each `FavoriteColors` option. Assuming that `Name` does not include NULL values, this count includes a count of NULL values in `FavoriteColors`.

SQL

```
SELECT
  FavoriteColors,
  COUNT(FavoriteColors) AS ColorPreference,
  COUNT(Name) AS People
FROM Sample.Person
GROUP BY FavoriteColors
```

This query returns the count of person rows for each `Home_State` value in `Sample.Person`.

SQL

```
SELECT
  Home_State,
  COUNT(*) AS AllPersons
FROM Sample.Person
GROUP BY Home_State
```

This query uses **%AFTERHAVING** to return the count of person rows, and the count of persons over 65, for each state with at least one person over 65.

SQL

```
SELECT
  Home_State,
  COUNT(Name) AS AllPersons,
  COUNT(Name %AFTERHAVING) AS Seniors
FROM Sample.Person
GROUP BY Home_State
HAVING Age > 65
ORDER BY Home_State
```

This query uses both the **%FOREACH** and **%AFTERHAVING** keywords. It uses **GROUP BY** to return one row per state and **HAVING** to filter only on people whose names start with "A", "M", or "W".

SQL

```
SELECT
  Home_State,
  COUNT(Name) AS NameCount,
  COUNT(Name %FOREACH(Home_State)) AS StateNameCount,
  COUNT(Name %AFTERHAVING) AS NameCountHaving,
  COUNT(Name %FOREACH(Home_State) %AFTERHAVING) AS StateNameCountHaving
FROM Sample.Person
GROUP BY Home_State
HAVING Name LIKE 'A%' OR Name LIKE 'M%' OR Name LIKE 'W%'
ORDER BY Home_State
```

Each state row contains these counts:

- **COUNT(Name)** — All people in the database. Assuming `Name` is required, this count is the same for all rows.

- **COUNT(Name %FOREACH(Home_State))** — All people in the state.

- **COUNT(Name %AFTERHAVING)** — All people in the database that meet the **HAVING** condition. Assuming Name is required, this number is the same for all rows.

- **COUNT(Name %FOREACH(Home_State) %AFTERHAVING)**: All people in the state that meet the **HAVING** condition.

Count Non-NULL Values in Combination of Columns

This query uses **COUNT** with a concatenation operator (||) to count the rows in which both the FavoriteColors and Home_State columns do not contain NULL values.

SQL

```
SELECT COUNT(FavoriteColors||Home_State) AS ColorState
FROM Sample.Person
```

Count Stream Column Values

You can use **COUNT(*expression*)** to count stream column values, with some restrictions:

- Column counts always include all non-NULL values, including duplicate values.

- You cannot specify a stream field in a **COUNT DISTINCT *expression*** clause. Attempting to do so results in an SQLCODE -37 error.

- You cannot specify a stream field in a **%FOREACH *column*** clause. Attempting to do so results in an SQLCODE -37 error.

This query shows a valid use of the **COUNT** function, where Title is a string field and Notes and Picture are stream fields:

SQL

```
SELECT DISTINCT Title,COUNT(Notes),COUNT(Picture %FOREACH(Title))
FROM Sample.Employee
```

These queries containing stream fields are *not* valid.

SQL

```
-- Invalid: DISTINCT keyword with stream field
SELECT Title,COUNT(DISTINCT Notes) FROM Sample.Employee
```

SQL

```
-- Invalid: %FOREACH col-list contains stream field
SELECT Title,COUNT(Notes %FOREACH(Picture))
FROM Sample.Employee
```

No Rows Returned in Count

These examples show what **COUNT** returns when the **SELECT** query selects no rows for the function to count. Depending on the query, **COUNT** returns either 0 or NULL.

If the **SELECT** *selectItem* list does not contain any references to columns in the **FROM** clause tables, other than columns supplied to aggregate functions, **COUNT** returns 0.

COUNT is the only aggregate function that returns 0. All other aggregate functions return NULL. The query returns a %ROWCOUNT of 1. Sample query:

SQL

```
SELECT
  COUNT(*) AS Recs, COUNT(Name) AS People,
  AVG(Age) AS AvgAge, MAX(Age) AS MaxAge,
  CURRENT_TIMESTAMP AS Now
FROM Sample.Employee
WHERE Name %STARTSWITH 'ZZZ'
```

If the **SELECT** *selectItem* list contains any direct reference to a column in a FROM clause table, or if TOP 0 is specified, **COUNT** returns NULL. The query returns a %ROWCOUNT of 0. Sample query:

SQL

```
SELECT
  COUNT(*) AS Recs,
  COUNT(Name) AS People,
  $LENGTH(Name) AS NameLen
FROM Sample.Employee WHERE Name %STARTSWITH 'ZZZ'
```

If no table is specified, **COUNT(*)** returns 1. The query returns a %ROWCOUNT of 1. Sample query:

SQL

```
SELECT COUNT(*) AS Recs
```

Security and Privileges

To use the **COUNT(*)** syntax, you must have table-level SELECT privilege for the specified table.

To use **COUNT(*expression*)** syntaxes, you must have either column-level SELECT privilege for the column specified by *expression* or table-level SELECT privilege for the specified table.

- To determine if you have SELECT privilege, use %CHECKPRIV.

- To determine if you have table-level SELECT privilege, use **$SYSTEM.SQL.Security.CheckPrivilege()**.

- To assign privileges, use GRANT.

Performance

To improve **COUNT** performance, consider defining these indexes:

- For the **COUNT(*)** syntax, define a bitmap extent index, if this index was not automatically defined when the table was created.

- For **COUNT(*expression*)** syntaxes, define a bitslice index for the column specified by *expression*. Query Plan optimization of **COUNT(*expression*)** automatically applies default collation to the column being counted.

Transaction Considerations

Like all aggregate functions, **COUNT** returns the current state of the data, including uncommitted changes, regardless of the current transaction's isolation level. **COUNT** follows this behavior:

- Counts inserted and updated records, even if those changes have not been committed and can be rolled back.

- Does not count deleted records, even if those deletions have not been committed and can be rolled back.

For more details, see SET TRANSACTION and START TRANSACTION.

See Also

- Aggregate Functions

- AVG
- SUM

%DLIST (SQL)

An aggregate function that creates an InterSystems IRIS list of values.

Synopsis

```
%DLIST([ALL | DISTINCT [BY(col-list)]] string-expr [%FOREACH(col-list)] [%AFTERHAVING])
```

Description

The **%DLIST** aggregate function returns an ObjectScript %List structure containing the values in the specified column as list elements.

A simple **%DLIST** (or **%DLIST ALL**) returns InterSystems IRIS list composed of all the non-NULL values for *string-expr* in the selected rows. Rows where *string-expr* is NULL are not included as elements in the list structure.

A **%DLIST DISTINCT** returns an InterSystems IRIS list composed of all the distinct (unique) non-NULL values for *string-expr* in the selected rows: %DLIST(DISTINCT col1). NULL is not included as an element in the %List structure. %DLIST(DISTINCT BY(col2) col1) returns a %List of elements including only those col1 field values in records where the col2 values are distinct (unique). Note however that the distinct col2 values may include a single NULL as a distinct value.

For further information about InterSystems IRIS list structures, see $LIST and related functions.

%DLIST and %SelectMode

You can use the %SelectMode property to specify the data display mode returned by **%DLIST**: 0=Logical (the default), 1=ODBC, 2=Display.

Note that **%DLIST** in ODBC mode separates column value lists with commas, and **$LISTTOSTRING** (by default) returns elements within a %List column value separated with commas.

%DLIST and ORDER BY

The **%DLIST** function combines the values of a table column from multiple rows into %List structured list of values. Because an **ORDER BY** clause is applied to the query result set after all aggregate fields are evaluated, **ORDER BY** cannot directly affect the sequence of values within this list. Under certain circumstances, **%DLIST** results may appear in sequential order, but this ordering should not be relied upon. The values listed within a given aggregate result value cannot be explicitly ordered.

Related Aggregate Functions

- **%DLIST** returns an InterSystems IRIS list of values.
- LIST returns a comma-separated list of values.
- JSON_ARRAYAGG returns a JSON array of values.
- XMLAGG returns a concatenated string of values.

Arguments

ALL

An optional argument specifying that **%DLIST** returns a list of all values for *string-expr*. This is the default if no keyword is specified.

DISTINCT

An optional DISTINCT clause that specifies that **%DLIST** returns a %List structured list containing only the unique *string-expr* values. DISTINCT can specify a BY(col-list) subclause, where *col-list* can be a single field or a comma-separated list of fields.

string-expr

An SQL expression that evaluates to a string. Usually the name of a column from the selected table.

%FOREACH(col-list)

An optional column name or a comma-separated list of column names. See SELECT for further information on %FOREACH.

%AFTERHAVING

An optional argument that applies the condition found in the HAVING clause.

Examples

The following example returns an InterSystems IRIS list of all of the values listed in the Home_State column of the Sample.Person table that start with the letter "A":

SQL

```
SELECT %DLIST(Home_State)
FROM Sample.Person
WHERE Home_State %STARTSWITH 'A'
```

Note that this InterSystems IRIS list contains elements with duplicate values.

The following example returns an InterSystems IRIS list of all of the distinct (unique) values listed in the Home_State column of the Sample.Person table that start with the letter "A":

SQL

```
SELECT %DLIST(DISTINCT Home_State)
FROM Sample.Person
WHERE Home_State %STARTSWITH 'A'
```

The following example creates an InterSystems IRIS list of all of the values found in the Home_City column for each of the states, and a count of these city values by state. Every Home_State row contains a list of all of the Home_City values for that state. These lists may include duplicate city names:

SQL

```
SELECT Home_State,
       %DLIST(Home_City) AS AllCities,
       COUNT(Home_City) AS CityCount
FROM Sample.Person
GROUP BY Home_State
```

Perhaps more useful would be a list of all of the *distinct* values found in the Home_City column for each of the states, as shown in the following example:

SQL

```
SELECT Home_State,
       %DLIST(DISTINCT Home_City) AS CitiesList,
       COUNT(DISTINCT Home_City) AS DistinctCities,
       COUNT(Home_City) AS TotalCities
FROM Sample.Person
GROUP BY Home_State
```

Note that this example returns integer counts of both the distinct city names and the total city names for each state.

The following example returns %List structures of Home_State values that begin with "A". It returns the following as %List elements: the distinct Home_State values (`DISTINCT Home_State`); the Home_State values corresponding to distinct Home_City values (`DISTINCT BY(Home_City) Home_State`), which may possibly including one unique NULL for Home_City; and all Home_State values:

SQL

```
SELECT %DLIST(DISTINCT Home_State) AS DistStates,
       %DLIST(DISTINCT BY(Home_City) Home_State) AS DistCityStates,
       %DLIST(Home_State) AS AllStates
FROM Sample.Person
WHERE Home_State %STARTSWITH 'A'
```

The following Dynamic SQL example uses the %SelectMode property to specify the ODBC display mode for the %List structure FavoriteColors date field. ODBC mode returns the value for each column as a comma-separated list, and the **$LISTTOSTRING** function specifies a different delimiter (in this example, ||) to separate the values from the different columns:

ObjectScript

```
  set myquery = "SELECT %DLIST(FavoriteColors) AS colors FROM Sample.Person WHERE Name %STARTSWITH 'A'"

  set tStatement = ##class(%SQL.Statement).%New()
  set tStatement.%SelectMode=1

  set qStatus = tStatement.%Prepare(myquery)
  if $$$ISERR(qStatus) {write "%Prepare failed:" do $SYSTEM.Status.DisplayError(qStatus) quit}

  set rset = tStatement.%Execute()
  if (rset.%SQLCODE '= 0) {write "%Execute failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message
 quit}

  while rset.%Next()
  {
    write $LISTTOSTRING(rset.colors,"||"),!
  }
  if (rset.%SQLCODE < 0) {write "%Next failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message
quit}

  write !,"End of data"
```

The following example uses the %AFTERHAVING keyword. It returns a row for each Home_State that contains at least one Name value that fulfills the HAVING clause condition (a name that begins with "M"). The first **%DLIST** function returns a list of all of the names for that state. The second **%DLIST** function returns a list containing only those names that fulfill the HAVING clause condition:

SQL

```
SELECT Home_State,
       %DLIST(Name) AS AllNames,
       %DLIST(Name %AFTERHAVING) AS HaveClauseNames
    FROM Sample.Person
    GROUP BY Home_State
    HAVING Name LIKE 'M%'
    ORDER BY Home_state
```

See Also

- Aggregate Functions overview

- SELECT

- $LIST function

- JSON_ARRAYAGG aggregate function

- LIST aggregate function

- XMLAGG aggregate function

JSON_ARRAYAGG (SQL)

An aggregate function that creates a JSON format array of values.

Synopsis

```
JSON_ARRAYAGG([ ALL | DISTINCT [BY(col-list)] ] string-expr
    [ %FOREACH(col-list) ] [ %AFTERHAVING ])
```

Arguments

Argument	Description
ALL	*Optional* — Specifies that **JSON_ARRAYAGG** returns a JSON array containing all values for *string-expr*. This is the default if no keyword is specified.
DISTINCT	*Optional* — A DISTINCT clause that specifies that **JSON_ARRAYAGG** returns a JSON array containing only the unique *string-expr* values. DISTINCT can specify a BY(col-list) subclause, where *col-list* can be a single field or a comma-separated list of fields.
string-expr	An SQL expression that evaluates to a string. Usually the name of a column from the selected table.
%FOREACH(*col-list*)	*Optional* — A column name or a comma-separated list of column names. See SELECT for further information on %FOREACH.
%AFTERHAVING	*Optional* — Applies the condition found in the HAVING clause.

Description

The **JSON_ARRAYAGG** aggregate function returns a JSON format array of the values in the specified column. For further details on JSON array format, refer to the JSON_ARRAY function.

A simple **JSON_ARRAYAGG** (or **JSON_ARRAYAGG ALL**) returns a JSON array containing all the values for *string-expr* in the selected rows. Rows where *string-expr* is the empty string ('') are represented by ("\u0000") in the array. Rows where *string-expr* is NULL are not included in the array. If there is only one *string-expr* value, and it is the empty string (''), **JSON_ARRAYAGG** returns the JSON array `["\u0000"]`. If all *string-expr* values are NULL, **JSON_ARRAYAGG** returns an empty JSON array `[]`.

A **JSON_ARRAYAGG DISTINCT** returns a JSON array composed of all the different (unique) values for *string-expr* in the selected rows: JSON_ARRAYAGG(DISTINCT col1). The NULL *string-expr* is not included in the JSON array. JSON_ARRAYAGG(DISTINCT BY(col2) col1) returns a JSON array containing only those col1 field values in records where the col2 values are distinct (unique). Note however that the distinct col2 values may include a single NULL as a distinct value.

The **JSON_ARRAYAGG** *string-expr* cannot be a stream field. Specifying a stream field results in an SQLCODE -37.

Data Values Containing Escaped Characters

- Double Quote: If a *string-expr* value contains a double quote character ("), **JSON_ARRAYAGG** represents this character using the literal escape sequence \ ".

- Backslash: If a *string-expr* value contains a backslash character (\), **JSON_ARRAYAGG** represents this character using the literal escape sequence \\.

- Single Quote: When a *string-expr* value contains a single quote as a literal character, InterSystems SQL requires that this character must be escaped by doubling it as two single quote characters (`''`. **JSON_ARRAYAGG** represents this character as a single quote character `'`.

Maximum JSON Array Size

The default **JSON_ARRAYAGG** return type is VARCHAR(8192). This length includes the JSON array formatting characters as well as the field data characters. If you anticipate the value returned will need to be longer than 8192, you can use the CAST function to specify a larger return value. For example, `CAST(JSON_ARRAYAGG(value)) AS VARCHAR(12000))`. If the actual JSON array returned is longer than the **JSON_ARRAYAGG** return type length, InterSystems IRIS truncates the JSON array at the return type length without issuing an error. Because truncating a JSON array removes its closing] character, this makes the return value invalid.

JSON_ARRAYAGG and %SelectMode

You can use the %SelectMode property to specify the data display values for the elements in the JSON array: 0=Logical (the default), 1=ODBC, 2=Display. If the *string-expr* contains a %List structure, the elements are represented in ODBC mode separated by a comma, and in Logical and Display mode with %List format characters represented by \ escape sequences. Refer to $ZCONVERT "Encoding Translation" for an table listing these JSON \ escape sequences.

JSON_ARRAYAGG and ORDER BY

The **JSON_ARRAYAGG** function combines the values of a table column from multiple rows into a JSON array of element values. Because an **ORDER BY** clause is applied to the query result set after all aggregate fields are evaluated, **ORDER BY** cannot directly affect the sequence of values within this list. Under certain circumstances, **JSON_ARRAYAGG** results may appear in sequential order, but this ordering should not be relied upon. The values listed within a given aggregate result value cannot be explicitly ordered.

Related Aggregate Functions

- LIST returns a comma-separated list of values.

- %DLIST returns an InterSystems IRIS list containing an element for each value.

- XMLAGG returns a concatenated string of values.

Examples

This example returns a JSON array of all values in the Home_State column of the Sample.Person table that start with the letter "A":

SQL

```
SELECT JSON_ARRAYAGG(Home_State)
FROM Sample.Person
WHERE Home_State %STARTSWITH 'A'
```

Note that this JSON array contains duplicate values.

The following example returns a host variable containing a JSON array of all of the distinct (unique) values in the Home_State column of the Sample.Person table that start with the letter "A":

SQL

```
SELECT DISTINCT JSON_ARRAYAGG(Home_State) AS DistinctStates
FROM Sample.Person
WHERE Home_State %STARTSWITH 'A'
```

The following example creates a JSON array of all of the values found in the Home_City column for each of the states, and a count of these city values by state. Every Home_State row contains a JSON array of all of the Home_City values for that state. These JSON arrays may include duplicate city names:

SQL

```
SELECT Home_State,
       COUNT(Home_City) AS CityCount,
       JSON_ARRAYAGG(Home_City) AS ArrayAllCities
FROM Sample.Person
GROUP BY Home_State
```

Perhaps more useful would be a JSON array of all of the *distinct* values found in the Home_City column for each of the states, as shown in the following example:

SQL

```
SELECT DISTINCT Home_State,
       COUNT(DISTINCT Home_City) AS DistCityCount,
       COUNT(Home_City) AS TotCityCount,
       JSON_ARRAYAGG(DISTINCT Home_City) AS ArrayDistCities
FROM Sample.Person GROUP BY Home_State
```

Note that this example returns integer counts of both the distinct city names and the total city names for each state.

The following Dynamic SQL example uses the %SelectMode property to specify the ODBC display mode for the JSON array of values returned by the DOB date field:

ObjectScript

```
  SET myquery = 2
  SET myquery(1) = "SELECT JSON_ARRAYAGG(DOB) AS DOBs "
  SET myquery(2) = "FROM Sample.Person WHERE Name %STARTSWITH 'A'"
  SET tStatement = ##class(%SQL.Statement).%New()
  SET tStatement.%SelectMode=1
  SET qStatus = tStatement.%Prepare(.myquery)
    IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET rset = tStatement.%Execute()
  DO rset.%Display()
  WRITE !,"End of data"
```

The following example uses the %FOREACH keyword. It returns a row for each distinct Home_State containing a JSON array of age values for that Home_State.

SQL

```
SELECT DISTINCT Home_State,
    JSON_ARRAYAGG(Age %FOREACH(Home_State))
FROM Sample.Person
WHERE Home_State %STARTSWITH 'M'
```

The following example uses the %AFTERHAVING keyword. It returns a row for each Home_State that contains at least one Name value that fulfills the HAVING clause condition (a name that begins with "M"). The first **JSON_ARRAYAGG** function returns a JSON array of all of the names for that state. The second **JSON_ARRAYAGG** function returns a JSON array containing only those names that fulfill the HAVING clause condition:

SQL

```
SELECT Home_State,
       JSON_ARRAYAGG(Name) AS AllNames,
       JSON_ARRAYAGG(Name %AFTERHAVING) AS HavingClauseNames
FROM Sample.Person GROUP BY Home_State
HAVING Name LIKE 'M%' ORDER BY Home_State
```

See Also

- Aggregate Functions overview
- JSON_ARRAY function
- IS JSON predicate condition
- LIST aggregate function
- %DLIST aggregate function
- XMLAGG aggregate function
- SELECT statement

LIST (SQL)

An aggregate function that creates a comma-separated list of values.

Synopsis

```
LIST([ ALL | DISTINCT [BY(col-list)] ] string-expr
    [ %FOREACH(col-list) ]
    [ %AFTERHAVING] )
```

Arguments

Argument	Description
ALL	*Optional* — Specifies that **LIST** returns a list of all values for *string-expr*. This is the default if no keyword is specified.
DISTINCT	*Optional* — A DISTINCT clause that specifies that **LIST** returns a list containing only the unique *string-expr* values. DISTINCT can specify a `BY(col-list)` subclause, where *col-list* can be a single field or a comma-separated list of fields.
string-expr	An SQL expression that evaluates to a string. Usually the name of a column from the selected table.
%FOREACH(*col-list*)	*Optional* — A column name or a comma-separated list of column names. See SELECT for further information on %FOREACH.
%AFTERHAVING	*Optional* — Applies the condition found in the HAVING clause.

Description

The **LIST** aggregate function returns a comma-separated list of the values in the specified column.

A simple **LIST** (or **LIST ALL**) returns a string containing a comma-separated list composed of all the values for *string-expr* in the selected rows. Rows where *string-expr* is the empty string (") are represented by a placeholder comma in the comma-separated list. Rows where *string-expr* is NULL are not included in the comma-separated list. If there is only one *string-expr* value, and it is the empty string ("), **LIST** returns the empty string.

A **LIST DISTINCT** returns a string containing a comma-separated list composed of all the distinct (unique) values for *string-expr* in the selected rows: `LIST(DISTINCT col1)`. The NULL *string-expr* is not included in the comma-separated list. `LIST(DISTINCT BY(col2) col1)` returns a comma-separated list containing only those col1 field values in records where the col2 values are distinct (unique). Note however that the distinct col2 values may include a single NULL as a distinct value.

Data Values Containing Commas

Because **LIST** uses commas to separate *string-expr* values, **LIST** should not be used for data values that contain commas. Use **%DLIST** or **JSON_ARRAYAGG** instead.

LIST and %SelectMode

You can use the %SelectMode property to specify the data display mode returned by LIST: 0=Logical (the default), 1=ODBC, 2=Display.

Note that **LIST** separates column values with commas, and ODBC mode separates elements within a %List column value with commas. Therefore, using ODBC mode when using **LIST** on a %List structure produces ambiguous results.

LIST and ORDER BY

The **LIST** function combines the values of a table column from multiple rows into a single comma-separated list of values. Because an **ORDER BY** clause is applied to the query result set after all aggregate fields are evaluated, **ORDER BY** cannot directly affect the sequence of values within this list. Under certain circumstances, **LIST** results may appear in sequential order, but this ordering should not be relied upon. The values listed within a given aggregate result value cannot be explicitly ordered.

Maximum LIST Size

Any **LIST** return value must be not longer than the maximum string length.

Related Aggregate Functions

- **LIST** returns a comma-separated list of values.

- %DLIST returns a list containing an element for each value.

- JSON_ARRAYAGG returns a JSON array of values.

- XMLAGG returns a concatenated string of values.

Examples

The following SQL example returns a host variable containing a comma-separated list of all of the values listed in the Home_State column of the Sample.Person table that start with the letter "A":

SQL

```
SELECT LIST(Home_State) AS StateList
FROM Sample.Person
WHERE Home_State %STARTSWITH 'A'
```

Note that this list contains duplicate values.

The following SQL example returns a host variable containing a comma-separated list of all of the distinct (unique) values listed in the Home_State column of the Sample.Person table that start with the letter "A":

SQL

```
SELECT LIST(DISTINCT Home_State) AS DistinctStates
FROM Sample.Person
WHERE Home_State %STARTSWITH 'A'
```

The following SQL example creates a comma-separated list of all of the values found in the Home_City column for each of the states, and a count of these city values by state. Every Home_State row contains a list of all of the Home_City values for that state. These lists may include duplicate city names:

SQL

```
SELECT Home_State,
       COUNT(Home_City) AS CityCount,
       LIST(Home_City) AS ListAllCities
FROM Sample.Person
GROUP BY Home_State
```

Perhaps more useful would be a comma-separated list of all of the *distinct* values found in the Home_City column for each of the states, as shown in the following example:

SQL

```
SELECT Home_State,
       COUNT(DISTINCT Home_City) AS DistCityCount,
       COUNT(Home_City) AS TotCityCount,
       LIST(DISTINCT Home_City) AS DistCitiesList
FROM Sample.Person
GROUP BY Home_State
```

Note that this example returns integer counts of both the distinct city names and the total city names for each state.

The following example returns lists of Home_State values that begin with "A". It returns the distinct Home_State values (`DISTINCT Home_State`); the Home_State values corresponding to distinct Home_City values (`DISTINCT BY(Home_City) Home_State`), which may possibly including one unique NULL for Home_City; and all Home_State values:

SQL

```
SELECT LIST(DISTINCT Home_State) AS DistStates,
       LIST(DISTINCT BY(Home_City) Home_State) AS DistCityStates,
       LIST(Home_State) AS AllStates
FROM Sample.Person
WHERE Home_State %STARTSWITH 'A'
```

The following Dynamic SQL example uses the %SelectMode property to specify the ODBC display mode for the list of values returned by the DOB date field:

ObjectScript

```
  SET myquery = "SELECT LIST(DOB) AS DOBs FROM Sample.Person WHERE Name %STARTSWITH 'A'"
  SET tStatement = ##class(%SQL.Statement).%New()
  SET tStatement.%SelectMode=1
  SET qStatus = tStatement.%Prepare(myquery)
    IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET rset = tStatement.%Execute()
  DO rset.%Display()
  WRITE !,"End of data"
```

The following example uses the %FOREACH keyword. It returns a row for each distinct Home_State containing a list of age values for that Home_State:

SQL

```
SELECT DISTINCT Home_State,
       LIST(Age %FOREACH(Home_State)) AgesForState
FROM Sample.Person WHERE Home_State %STARTSWITH 'M'
```

The following example uses the %AFTERHAVING keyword. It returns a row for each Home_State that contains at least one Name value that fulfills the HAVING clause condition (a name that begins with "M"). The first **LIST** function returns a list of all of the names for that state. The second **LIST** function returns a list containing only those names that fulfill the HAVING clause condition:

SQL

```
SELECT Home_State,
       LIST(Name) AS AllNames,
       LIST(Name %AFTERHAVING) AS HavingClauseNames
FROM Sample.Person
GROUP BY Home_State
HAVING Name LIKE 'M%'
ORDER BY Home_State
```

See Also

- Aggregate Functions overview

- %DLIST aggregate function

- **JSON_ARRAYAGG** aggregate function

- **XMLAGG** aggregate function

- **SELECT** statement

MAX (SQL)

An aggregate function that returns the maximum data value in a specified column.

Synopsis

```
MAX([ALL | DISTINCT [BY(col-list)]] expression [%FOREACH(col-list)] [%AFTERHAVING])
```

Note: **MAX** can be specified as an aggregate function or as a window function. This reference page describes the use of **MAX** as an aggregate function. **MAX** as a window function is described in Overview of Window Functions.

Description

The **MAX** aggregate function returns the largest (maximum) of the values of *expression*. Commonly, *expression* is the name of a field, (or an expression containing one or more field names) in the multiple rows returned by a query.

MAX can be used in a SELECT query or subquery that references either a table or a view. **MAX** can appear in a SELECT list or HAVING clause alongside ordinary field values.

MAX cannot be used in a WHERE clause. **MAX** cannot be used in the ON clause of a JOIN, unless the SELECT is a subquery.

Like most other aggregate functions, **MAX** cannot be applied to a stream field. Attempting to do so generates an SQLCODE -37 error.

Unlike most other aggregate functions, the ALL and DISTINCT keywords, including MAX(DISTINCT BY(col2) col1), perform no operation in **MAX**. They are provided for SQL–92 compatibility.

Data Values

The specified field used by **MAX** can be numeric or nonnumeric. For a numeric data type field, maximum is defined as highest in numeric value; thus -3 is higher than -7. For a non-numeric data type field, maximum is defined as highest in string collation sequence; thus '-7' is higher than '-3'.

An empty string ('') value is treated as CHAR(0).

A predicate uses the collation type defined for the field. By default, string data type fields are defined with SQLUPPER collation, which is not case-sensitive. You can define the string collation default for the current namespace and specify a non-default field collation type when defining a field/property.

When the field's defined collation type is SQLUPPER, **MAX** returns strings in all uppercase letters. Thus SELECT MAX(Name) returns 'ZWIG', regardless of the original lettercase of the data. But because comparisons are performed using uppercase collation, the clause HAVING Name=MAX(Name) selects rows with the Name value 'Zwig', 'ZWIG', and 'zwig'.

For numeric values, the scale returned is the same as the *expression* scale.

NULL values in data fields are ignored when deriving a **MAX** aggregate function value. If no rows are returned by the query, or the data field value for all rows returned is NULL, **MAX** returns NULL.

Arguments

ALL

An optional argument that applies the aggregate function to all values. ALL has no effect on the value returned by **MAX**. It is provided for SQL-92 compatibility.

DISTINCT

An optional DISTINCT clause that specifies that each unique value is considered. DISTINCT has no effect on the value returned by **MAX**. It is provided for SQL-92 compatibility.

expression

Any valid expression. Usually the name of a column that contains the values from which the maximum value is to be returned.

%FOREACH(col-list)

An optional column name or a comma-separated list of column names. See SELECT for further information on %FOREACH.

%AFTERHAVING

An optional argument that applies the condition found in the HAVING clause.

MAX returns the same data type as *expression*.

Note: **MAX** can be specified as an aggregate function or as a window function. This reference page describes the use of **MAX** as an aggregate function. **MAX** as a window function is described in Overview of Window Functions.

Changes Made During the Current Transaction

Like all aggregate functions, **MAX** always returns the current state of the data, including uncommitted changes, regardless of the current transaction's isolation level. For further details, refer to SET TRANSACTION and START TRANSACTION.

Examples

The following query returns the highest (maximum) salary in the Sample.Employee database:

SQL

```
SELECT '$' || MAX(Salary) As TopSalary
    FROM Sample.Employee
```

The following query returns one row for each state that contains at least one employee with a salary smaller than $25,000. Using the %AFTERHAVING keyword, each row returns the maximum employee salary smaller than $25,000. Each row also returns the minimum salary and the maximum salary for all employees in that state:

SQL

```
SELECT Home_State,
       '$' || MAX(Salary %AFTERHAVING) AS MaxSalaryBelow25K,
       '$' || MIN(Salary) AS MinSalary,
       '$' || MAX(Salary) AS MaxSalary
         FROM Sample.Employee
     GROUP BY Home_State
     HAVING Salary < 25000
     ORDER BY Home_State
```

The following query returns the lowest (minimum) and highest (maximum) name in collation sequence found in the Sample.Employee database:

SQL

```
SELECT Name,MIN(Name),MAX(Name)
    FROM Sample.Employee
```

Note that **MIN** and **MAX** convert Name values to uppercase before comparison.

The following query returns the highest (maximum) salary for an employee whose Home_State is 'VT' in the Sample.Employee database:

SQL

```
SELECT MAX(Salary)
     FROM Sample.Employee
     WHERE Home_State = 'VT'
```

The following query returns the number of employees and the highest (maximum) employee salary for each Home_State in the Sample.Employee database:

SQL

```
SELECT Home_State,
     COUNT(Home_State) As NumEmployees,
     MAX(Salary) As TopSalary
     FROM Sample.Employee
     GROUP BY Home_State
     ORDER BY TopSalary
```

See Also

- Aggregate Functions overview

- MIN aggregate function

MIN (SQL)

An aggregate function that returns the minimum data value in a specified column.

Synopsis

```
MIN([ALL | DISTINCT [BY(col-list)]] expression [%FOREACH(col-list)] [%AFTERHAVING])
```

Note: **MIN** can be specified as an aggregate function or as a window function. This reference page describes the use of **MIN** as an aggregate function. **MIN** as a window function is described in Overview of Window Functions.

Description

The **MIN** aggregate function returns the smallest (minimum) of the values of *expression*. Commonly, *expression* is the name of a field, (or an expression containing one or more field names) in the multiple rows returned by a query.

MIN can be used in a SELECT query or subquery that references either a table or a view. **MIN** can appear in a SELECT list or HAVING clause alongside ordinary field values.

MIN cannot be used in a WHERE clause. **MIN** cannot be used in the ON clause of a JOIN, unless the SELECT is a subquery.

Like most other aggregate functions, **MIN** cannot be applied to a stream field. Attempting to do so generates an SQLCODE -37 error.

Unlike most other aggregate functions, the ALL and DISTINCT keywords, including MIN(DISTINCT BY(col2) col1), perform no operation in **MIN**. They are provided for SQL–92 compatibility.

Data Values

The specified field used by **MIN** can be numeric or nonnumeric. For a numeric data type field, minimum is defined as lowest in numeric value; thus -7 is lower than -3. For a non-numeric data type field, minimum is defined as lowest in string collation sequence; thus '-3' is lower than '-7'.

An empty string ('') value is treated as CHAR(0).

A predicate uses the collation type defined for the field. By default, string data type fields are defined with SQLUPPER collation, which is not case-sensitive. You can define the string collation default for the current namespace and specify a non-default field collation type when defining a field/property.

When the field's defined collation type is SQLUPPER, **MIN** returns strings in all uppercase letters. Thus SELECT MIN(Name) returns 'AARON', regardless of the original lettercase of the data. But because comparisons are performed using uppercase collation, the clause HAVING Name=MIN(Name) selects rows with the Name value 'Aaron', 'AARON', and 'aaron'.

For numeric values, the scale returned is the same as the *expression* scale.

NULL values in data fields are ignored when deriving a **MIN** aggregate function value. If no rows are returned by the query, or the data field value for all rows returned is NULL, **MIN** returns NULL.

Arguments

ALL

An optional argument that applies the aggregate function to all values. ALL has no effect on the value returned by **MIN**. It is provided for SQL-92 compatibility.

DISTINCT

An optional argument specifying that each unique value is considered. DISTINCT has no effect on the value returned by **MIN**. It is provided for SQL-92 compatibility.

expression

Any valid expression. Usually the name of a column that contains the values from which the minimum value is to be returned.

%FOREACH(col-list)

An optional column name or a comma-separated list of column names. See SELECT for further information on %FOREACH.

%AFTERHAVING

An optional argument that applies the condition found in the HAVING clause.

MIN returns the same data type as *expression*.

Note: **MIN** can be specified as an aggregate function or as a window function. This reference page describes the use of **MIN** as an aggregate function. **MIN** as a window function is described in Overview of Window Functions.

Changes Made During the Current Transaction

Like all aggregate functions, **MIN** always returns the current state of the data, including uncommitted changes, regardless of the current transaction's isolation level. For further details, refer to SET TRANSACTION and START TRANSACTION.

Examples

In the following examples a dollar sign ($) is concatenated to Salary amounts.

The following query returns the lowest (minimum) salary in the Sample.Employee database:

SQL

```
SELECT '$' || MIN(Salary) AS LowSalary
    FROM Sample.Employee
```

The following query returns one row for each state that contains at least one employee with a salary larger than $75,000. Using the %AFTERHAVING keyword, each row returns the minimum employee salary larger than $75,000. Each row also returns the minimum salary and the maximum salary for all employees in that state:

SQL

```
SELECT Home_State,
       '$' || MIN(Salary %AFTERHAVING) AS MinSalaryAbove75K,
       '$' || MIN(Salary) AS MinSalary,
       '$' || MAX(Salary) AS MaxSalary
        FROM Sample.Employee
    GROUP BY Home_State
    HAVING Salary > 75000
    ORDER BY MinSalaryAbove75K
```

The following query returns the lowest (minimum) and highest (maximum) name in collation sequence found in the Sample.Employee database:

SQL

```
SELECT Name,MIN(Name),MAX(Name)
    FROM Sample.Employee
```

Note that **MIN** and **MAX** convert Name values to uppercase before comparison.

The following query returns the lowest (minimum) salary for an employee whose Home_State is 'VT' in the Sample.Employee database:

SQL

```
SELECT MIN(Salary)
     FROM Sample.Employee
     WHERE Home_State = 'VT'
```

The following query returns the number of employees and the lowest (minimum) employee salary for each Home_State in the Sample.Employee database:

SQL

```
SELECT Home_State,
     COUNT(Home_State) As NumEmployees,
     MIN(Salary) As LowSalary
     FROM Sample.Employee
     GROUP BY Home_State
     ORDER BY LowSalary
```

See Also

- Aggregate Functions overview

- MAX aggregate function

STDDEV, STDDEV_SAMP, STDDEV_POP (SQL)

Aggregate functions that return the statistical standard deviation of a data set.

Synopsis

```
STDDEV([ ALL | DISTINCT [BY(col-list)] ] expression
    [ %FOREACH(col-list) ]
    [ %AFTERHAVING ])

STDDEV_SAMP([ ALL | DISTINCT [BY(col-list)] ] expression
    [ %FOREACH(col-list) ]
    [ %AFTERHAVING ])

STDDEV_POP([ALL | DISTINCT [BY(col-list)]] expression [%FOREACH(col-list)] [%AFTERHAVING])
```

Description

These three standard deviation aggregate functions return the statistical standard deviation of the distribution of the values of *expression*, after discarding NULL values. That is, the amount of standard deviation from the mean value of the data set, expressed as a positive number. The larger the return value, the more variation there is within the data set of values.

The **STDDEV**, **STDDEV_SAMP** (sample), and **STDDEV_POP** (population) functions are derived from the corresponding variance aggregate functions:

STDDEV	VARIANCE
STDDEV_SAMP	VAR_SAMP
STDDEV_POP	VAR_POP

The standard deviation is the square root of the corresponding variance value. Refer to these variance aggregate functions for further details.

These standard deviation functions can be used in a SELECT query or subquery that references either a table or a view. They can appear in a **SELECT** list or **HAVING** clause alongside ordinary field values.

These standard deviation functions cannot be used in a **WHERE** clause. They cannot be used in the **ON** clause of a **JOIN**, unless the **SELECT** is a subquery.

These standard deviation functions return a value of data type NUMERIC with a precision of 36 and a scale of 17, unless *expression* is data type DOUBLE in which case it returns data type DOUBLE.

These functions are normally applied to a field or expression that has a numeric value. They evaluate nonnumeric values, including the empty string (''), as zero (0).

These standard deviation functions ignore NULL values in data fields. If no rows are returned by the query, or the data field value for all rows returned is NULL, they return NULL.

The standard deviation functions, like all aggregate functions, can take an optional DISTINCT clause. STDDEV(DISTINCT col1) returns the standard deviation of those col1 field values that are distinct (unique). STDDEV(DISTINCT BY(col2) col1) returns the standard deviation of the col1 field values in records where the col2 values are distinct (unique). Note however that the distinct col2 values may include a single NULL as a distinct value.

Arguments

ALL

An optional argument specifying that standard deviation functions return the standard deviation of all values for *expression*. This is the default if no keyword is specified.

DISTINCT

An optional DISTINCT clause that specifies that standard deviation functions return the standard deviation of the distinct (unique) *expression* values. DISTINCT can specify a `BY(col-list)` subclause, where *col-list* can be a single field or a comma-separated list of fields.

expression

Any valid expression. Usually the name of a column that contains the data values to be analyzed for standard deviation.

%FOREACH(col-list)

An optional column name or a comma-separated list of column names. See SELECT for further information on %FOREACH.

%AFTERHAVING

An optional argument that applies the condition found in the HAVING clause.

Changes Made During the Current Transaction

Like all aggregate functions, standard deviation functions always returns the current state of the data, including uncommitted changes, regardless of the current transaction's isolation level. For further details, refer to SET TRANSACTION and START TRANSACTION.

Examples

The following example uses **STDDEV** to return the standard deviation in the ages of the employees in Sample.Employee, and the standard deviation in the distinct ages represented by one or more employees:

```
SELECT STDDEV(Age) AS AgeSD,STDDEV(DISTINCT Age) AS PerAgeSD
    FROM Sample.Employee
```

The following example uses **STDDEV_POP** to return the population standard deviation in the ages of the employees in Sample.Employee, and the standard deviation in the distinct ages represented by one or more employees:

```
SELECT STDDEV_POP(Age) AS AgePopSD,STDDEV_POP(DISTINCT Age) AS PerAgePopSD
    FROM Sample.Employee
```

See Also

- Aggregate Functions overview

- VARIANCE, VAR_SAMP, VAR_POP aggregate functions

- AVG aggregate function

- COUNT aggregate function

SUM (SQL)

An aggregate function that returns the sum of the values of a specified column.

Synopsis

```
SUM([ALL | DISTINCT [BY(col-list)]] expression [%FOREACH(col-list)] [%AFTERHAVING])
```

Description

The **SUM** aggregate function returns the sum of the values of *expression*. Commonly, *expression* is the name of a field (or an expression containing one or more field names) in the multiple rows returned by a query.

SUM can be used in a SELECT query or subquery that references either a table or a view. **SUM** can appear in a **SELECT** list or **HAVING** clause alongside ordinary field values.

SUM cannot be used in a **WHERE** clause. **SUM** cannot be used in the **ON** clause of a **JOIN**, unless the **SELECT** is a subquery.

SUM, like all aggregate functions, can take an optional DISTINCT clause. SUM(DISTINCT col1) totals only those col1 field values that are distinct (unique). SUM(DISTINCT BY(col2) col1) totals only those col1 field values in records where the col2 values are distinct (unique). Note however that the distinct col2 values may include a single NULL as a distinct value.

Data Values

SUM returns data type INTEGER for an *expression* with data type INT, SMALLINT, or TINYINT. **SUM** returns data type BIGINT for an *expression* with data type BIGINT. **SUM** returns data type DOUBLE for an *expression* with data type DOUBLE. For all other numeric data types, **SUM** returns data type NUMERIC.

SUM returns a value with a precision of 18. The scale of the returned value is the same as the *expression* scale, with the following exception: if *expression* is a numeric value with data type VARCHAR or VARBINARY, the scale of the returned value is 8.

By default, aggregate functions use Logical (internal) data values, rather than Display values.

SUM is normally applied to a field or expression that has a numeric value. Because only minimal type checking is performed, it is possible (though rarely meaningful) to invoke it for nonnumeric fields. **SUM** evaluates nonnumeric values, including the empty string (''), as zero (0). If *expression* is data type VARCHAR, the return value to ODBC or JDBC is of data type DOUBLE.

NULL values in data fields are ignored when deriving a **SUM** aggregate function value. If no rows are returned by the query, or the data field value for all rows returned is NULL, **SUM** returns NULL.

Optimization

SQL optimization of a **SUM** calculation can use a bitslice index, if this index is defined for the field.

Arguments

ALL

An optional argument specifying that **SUM** return the sum of all values for *expression*. This is the default if no keyword is specified.

DISTINCT

An optional DISTINCT clause that specifies that **SUM** return the sum of the distinct (unique) values for *expression*. DISTINCT can specify a BY(col-list) subclause, where *col-list* can be a single field or a comma-separated list of fields.

expression

Any valid expression. Usually the name of a column that contains the data values to be summed.

%FOREACH(col-list)

An optional column name or a comma-separated list of column names. See SELECT for further information on %FOREACH.

%AFTERHAVING

An optional argument that applies the condition found in the HAVING clause.

SUM returns the same data type as *expression*, with the following exception: TINYINT, SMALLINT and INTEGER are all returned as data type INTEGER.

Note: **SUM** can be specified as an aggregate function or as a window function. This reference page describes the use of **SUM** as an aggregate function. **SUM** as a window function is described in Overview of Window Functions.

Changes Made During the Current Transaction

Like all aggregate functions, **SUM** always returns the current state of the data, including uncommitted changes, regardless of the current transaction's isolation level. For further details, refer to SET TRANSACTION and START TRANSACTION.

Examples

In the following examples a dollar sign ($) is concatenated to Salary amounts.

The following query returns the sum of the salaries of all employees in the Sample.Employee database:

SQL

```
SELECT '$' || SUM(Salary) AS Total_Payroll
    FROM Sample.Employee
```

The following query uses %AFTERHAVING to return the sum of all salaries and the sum of salaries over $80,000 for each state in which there is at least one person with a salary > $80,000:

SQL

```
SELECT Home_State,
       '$' || SUM(Salary) AS Total_Payroll,
       '$' || SUM(Salary %AFTERHAVING) AS Exec_Payroll
    FROM Sample.Employee
    GROUP BY Home_State
    HAVING Salary > 80000
    ORDER BY Home_State
```

The following query returns the sum and the average of the salaries for each job title in the Sample.Employee database:

SQL

```
SELECT Title,
       '$' || SUM(Salary) AS Total,
       '$' || AVG(Salary) AS Average
    FROM Sample.Employee
    GROUP BY Title
    ORDER BY Average
```

The following query shows **SUM** used with an arithmetic expression. For each job title in the Sample.Employee database it returns the sum of the current salaries and the sum of the salaries with a 10% increase in pay:

SQL

```
SELECT Title,
       '$' || SUM(Salary) AS BeforeRaises,
       '$' || SUM(Salary * 1.1) AS AfterRaises
    FROM Sample.Employee
    GROUP BY Title
    ORDER BY Title
```

The following query shows **SUM** used with a logical expression using the **CASE** statement. It counts all of the salaried employees, and uses **SUM** to count all of the salaried employees earning $90,000 or more.

SQL

```
SELECT COUNT(Salary) As AllPaid,
       SUM(CASE WHEN (Salary >= 90000)
           THEN 1 ELSE 0 END) As TopPaid
       FROM Sample.Employee
```

See Also

- Aggregate Functions overview

- AVG aggregate function

- COUNT aggregate function

VARIANCE, VAR_SAMP, VAR_POP (SQL)

Aggregate functions that return the statistical variance of a data set.

Synopsis

```
VARIANCE([ ALL | DISTINCT [BY(col-list)] ] expression
    [ %FOREACH(col-list) ] [ %AFTERHAVING ])

VAR_SAMP([ ALL | DISTINCT [BY(col-list)] ] expression
    [ %FOREACH(col-list) ] [ %AFTERHAVING ])

VAR_POP([ALL | DISTINCT [BY(col-list)]] expression [%FOREACH(col-list)] [%AFTERHAVING])
```

Description

These three variance aggregate functions return the statistical variance of the values of *expression*, after discarding NULL values. That is, the amount of variation from the mean value of the data set, expressed as a positive number. The larger the return value, the more variation there is within the data set of values. InterSystems SQL also supplies aggregate functions to return the standard deviation corresponding to each of these variance functions.

There are slight variations in how this statistical variation is derived:

- **VARIANCE**: Returns 0 if all of the values in the data set have the same value (no variability). Returns 0 if the data set consists of only one value (no possible variability). Returns NULL if the data set has no values.

  The **VARIANCE** calculation is:

  $$\frac{(SUM(expression**2) * COUNT(expression)) - SUM(expression**2)}{COUNT(expression) * (COUNT(expression) - 1)}$$

- **VAR_SAMP**: Sample variance. Returns 0 if all of the values in the data set have the same value (no variability). Returns NULL if the data set consists of only one value (no possible variability). Returns NULL if the data set has no values. Uses the same variant calculation as **VARIANCE**.

- **VAR_POP**: Population variance. Returns 0 if all of the values in the data set have the same value (no variability). Returns 0 if the data set consists of only one value (no possible variability). Returns NULL if the data set has no values.

  The **VAR_POP** calculation is:

  $$\frac{(SUM(expression**2) * COUNT(expression)) - (SUM(expression) **2)}{(COUNT(expression) **2)}$$

These variance aggregate functions can be used in a SELECT query or subquery that references either a table or a view. They can appear in a **SELECT** list or **HAVING** clause alongside ordinary field values.

These variance aggregate functions cannot be used in a **WHERE** clause. They cannot be used in the **ON** clause of a **JOIN**, unless the **SELECT** is a subquery.

These variance aggregate functions return a value of data type NUMERIC with a precision of 36 and a scale of 17, unless *expression* is data type DOUBLE in which case the function returns data type DOUBLE.

These variance aggregate functions are normally applied to a field or expression that has a numeric value. They evaluate nonnumeric values, including the empty string (''), as zero (0).

These variance aggregate functions ignore NULL values in data fields. If no rows are returned by the query, or the data field value for all rows returned is NULL, they return NULL.

The statistical variance functions, like all aggregate functions, can take an optional DISTINCT clause. `VARIANCE(DISTINCT col1)` returns the variance of those col1 field values that are distinct (unique). `VARIANCE(DISTINCT BY(col2)`

col1) returns the variance of the col1 field values in records where the col2 values are distinct (unique). Note however that the distinct col2 values may include a single NULL as a distinct value.

Arguments

ALL

An optional argument specifying that statistical variance functions return the variance of all values for *expression*. This is the default if no keyword is specified.

DISTINCT

An optional DISTINCT clause that specifies that statistical variance functions return the variance of the distinct (unique) *expression* values. DISTINCT can specify a BY(col-list) subclause, where *col-list* can be a single field or a comma-separated list of fields.

expression

Any valid expression. Usually the name of a column that contains the data values to be analyzed for variance.

%FOREACH(col-list)

An optional column name or a comma-separated list of column names. See SELECT for further information on %FOREACH.

%AFTERHAVING

An optional argument that applies the condition found in the HAVING clause.

Changes Made During the Current Transaction

Like all aggregate functions, the variance functions always returns the current state of the data, including uncommitted changes, regardless of the current transaction's isolation level. For further details, refer to SET TRANSACTION and START TRANSACTION.

Examples

The following example uses **VARIANCE** to return the variance in the ages of the employees in Sample.Employee, and the variance in the distinct ages represented by one or more employees:

```
SELECT VARIANCE(Age) AS AgeVar,VARIANCE(DISTINCT Age) AS PerAgeVar
    FROM Sample.Employee
```

The following example uses **VAR_POP** to return the population variance in the ages of the employees in Sample.Employee, and the variance in the distinct ages represented by one or more employees:

```
SELECT VAR_POP(Age) AS AgePopVar,VAR_POP(DISTINCT Age) AS PerAgePopVar
    FROM Sample.Employee
```

See Also

- Aggregate Functions overview

- AVG aggregate function

- COUNT aggregate function

- STDDEV, STDDEV_SAMP, STDDEV_POP aggregate functions

XMLAGG (SQL)

An aggregate function that creates a concatenated string of values.

Synopsis

```
XMLAGG([ ALL | DISTINCT [BY(col-list)] ] string-expr
    [ %FOREACH(col-list) ] [ %AFTERHAVING ])
```

Arguments

Argument	Description
ALL	*Optional* — Specifies that **XMLAGG** returns a concatenated string of all values for *string-expr*. This is the default if no keyword is specified.
DISTINCT	*Optional* — A DISTINCT clause that specifies that **XMLAGG** returns a concatenated string containing only the unique *string-expr* values. DISTINCT can specify a BY(col-list) subclause, where *col-list* can be a single field or a comma-separated list of fields.
string-expr	An SQL expression that evaluates to a string. Commonly this is the name of a column from which to retrieve data.
%FOREACH(*col-list*)	*Optional* — A column name or a comma-separated list of column names. See SELECT for further information on %FOREACH.
%AFTERHAVING	*Optional* — Applies the condition found in the HAVING clause.

Description

The **XMLAGG** aggregate function returns a concatenated string of all values from *string-expr*. The return value is of data type VARCHAR.

- A simple **XMLAGG** (or **XMLAGG ALL**) returns a string containing a concatenated string composed of all the values for *string-expr* in the selected rows. Rows where *string-expr* is NULL are ignored.

  The following two examples both return the same single value, a concatenated string of all of the values listed in the Home_State column of the Sample.Person table.

  SQL

  ```
  SELECT XMLAGG(Home_State) AS All_State_Values
  FROM Sample.Person
  ```

  SQL

  ```
  SELECT XMLAGG(ALL Home_State) AS ALL_State_Values
  FROM Sample.Person
  ```

  Note that this concatenated string contains duplicate values.

- An **XMLAGG DISTINCT** returns a concatenated string composed of all the distinct (unique) values for *string-expr* in the selected rows: XMLAGG(DISTINCT col1). Rows where *string-expr* is NULL are ignored. XMLAGG(DISTINCT BY(col2) col1) returns a concatenated string containing only those col1 field values in records where the col2 values are distinct (unique). Note however that the distinct col2 values may include a single NULL as a distinct value.

Rows where *string-expr* is NULL are omitted from the return value. Rows where *string-expr* is the empty string (") are omitted from the return value if at least one non-empty string value is returned. If the only non-NULL *string-expr* values are the empty string ("), the return value is a single empty string.

XMLAGG does not support data stream fields. Specifying a stream field for *string-expr* results in an SQLCODE -37.

XML and XMLAGG

One common use of **XMLAGG** is to tag each data item from a column. This is done by combining **XMLAGG** and **XMLELEMENT** as shown in the following example:

SQL

```
SELECT XMLAGG(XMLELEMENT("para",Home_State))
FROM Sample.Person
```

This results in an output string such as the following:

```
<para>LA</para><para>MN</para><para>LA</para><para>NH</para><para>ME</para>...
```

XMLAGG and ORDER BY

The **XMLAGG** function concatenates values of a table column from multiple rows into a single string. Because an **ORDER BY** clause is applied to the query result set after all aggregate fields are evaluated, **ORDER BY** cannot directly affect the sequence of values within this string. Under certain circumstances, **XMLAGG** results may appear in sequential order, but this ordering should not be relied upon. The values listed within a given aggregate result value cannot be explicitly ordered.

Related Aggregate Functions

- **XMLAGG** returns a string of concatenated values.

- LIST returns a comma-separated list of values.

- %DLIST returns an InterSystems IRIS list containing an element for each value.

- JSON_ARRAYAGG returns a JSON array of values.

Examples

The following example creates a concatenated string of all of the distinct values found in the FavoriteColors column of the Sample.Person table. Thus every row has the same value for the All_Colors column. Note that while some rows have a NULL value for FavoriteColors, this value is not included in the concatenated string. Data values are returned in internal format.

SQL

```
SELECT Name,FavoriteColors,
   XMLAGG(DISTINCT FavoriteColors) AS All_Colors_In_Table
FROM Sample.Person
ORDER BY FavoriteColors
```

The following example returns concatenated strings of Home_State values that begin with "A". It returns the distinct Home_State values (DISTINCT Home_State); the Home_State values corresponding to distinct Home_City values (DISTINCT BY(Home_City) Home_State), which may possibly including one unique NULL for Home_City; and all Home_State values:

SQL

```
SELECT XMLAGG(DISTINCT Home_State) AS DistStates,
       XMLAGG(DISTINCT BY(Home_City) Home_State) AS DistCityStates,
       XMLAGG(Home_State) AS AllStates
FROM Sample.Person
WHERE Home_State %STARTSWITH 'A'
```

The following example creates a concatenated string of all of the distinct values found in the Home_City column for each of the states. Every row from the same state contains a list of all of the distinct city values for that state:

SQL

```
SELECT Home_State, Home_City,
    XMLAGG(DISTINCT Home_City %FOREACH(Home_State)) AS All_Cities_In_State
FROM Sample.Person
ORDER BY Home_State
```

The following example uses the %AFTERHAVING keyword. It returns a row for each Home_State that contains at least one Name value that fulfills the HAVING clause condition (a name that begins with either "C" or "K"). The first **XMLAGG** function returns a concatenated string consisting of all of the names for that state. The second **XMLAGG** function returns a concatenated string consisting of only those names that fulfill the HAVING clause condition:

SQL

```
SELECT Home_State,
       XMLAGG(Name) AS AllNames,
       XMLAGG(Name %AFTERHAVING) AS HaveClauseNames
    FROM Sample.Person
    GROUP BY Home_State
    HAVING Name LIKE 'C%' OR Name LIKE 'K%'
    ORDER BY Home_state
```

For the following examples, suppose we have the following table, AutoClub:

Name	Make	Model	Year
Smith,Joe	Pontiac	Firebird	1971
Smith,Joe	Saturn	SW2	1997
Smith,Joe	Pontiac	Bonneville	1999
Jones,Scott	Ford	Mustang	1966
Jones,Scott	Mazda	Miata	2000

The query:

SQL

```
SELECT DISTINCT Name, XMLAGG(Make) AS String_Of_Makes
FROM AutoClub WHERE Name = 'Smith,Joe'
```

returns:

Name	String_Of_Makes
Smith,Joe	PontiacSaturnPontiac

The query:

SQL

```
SELECT DISTINCT Name, XMLAGG(DISTINCT Make) AS String_Of_Makes
FROM AutoClub WHERE Name = 'Smith,Joe'
```

returns:

Name	String_Of_Makes
Smith,Joe	PontiacSaturn

See Also

- Aggregate Functions overview

- %DLIST aggregate function

- JSON_ARRAYAGG aggregate function

- LIST aggregate function

- XMLELEMENT function

- SELECT statement

SQL Window Functions

Overview of Window Functions

Functions that specify a per-row "window frame" for calculating aggregates and ranking.

Window Functions and Aggregate Functions

A window function operates on the rows selected by a SELECT query after the WHERE, GROUP BY, and HAVING clauses have been applied.

A window function combines the values of a field (or fields) from a group of rows and return a value for each row in a generated column in the result set.

While window functions are like aggregate functions in that they combine results from multiple rows, they are distinct from aggregates in that they do not combine the rows themselves. However, the aggregate functions AVG(), MIN(), MAX(), and SUM() can also be invoked as window functions. Within this context, each row receives the result of calling the function on the group of rows in its corresponding window frame.

Window Functions Syntax

A window function is specified as a *select-item* in a SELECT query. A window function can also be specified in the ORDER BY clause of a SELECT query.

A window function performs a task in relation to a per-row window specified by a PARTITION BY clause, an ORDER BY clause, and a ROWS clause, and returns a value for each row. All three of these clauses are optional, but if specified must be specified in the orders shown in the following syntax:

```
window-function() OVER ( [ PARTITION BY partfield ]    [ ORDER BY orderfield ]    [ ROWS framestart
 ] | [ ROWS BETWEEN framestart AND frameend ] )
```

where *framestart* and *frameend* can be:

```
UNBOUNDED PRECEDING | offset PRECEDING | CURRENT ROW | UNBOUNDED FOLLOWING | offset FOLLOWING
```

Keywords and window function names are not case-sensitive.

A Simple Example

CityTable contains rows with the following values:

Name	City
Able	New York
Betty	Boston
Charlie	Paris
Davis	Boston
Eve	Paris
Francis	Paris
George	London
Beatrix	Paris

The ROW_NUMBER() window function assigns a unique sequential integer to each row based on the specified window.

```
SELECT Name,City,ROW_NUMBER() OVER (PARTITION BY City) FROM CityTable
```

This example partitions the rows by the City value and returns the following:

Name	City	Window_3
Able	New York	1
Betty	Boston	1
Charlie	Paris	1
Davis	Boston	2
Eve	Paris	2
Francis	Paris	3
George	London	1
Beatrix	Paris	4

```
SELECT Name,City,ROW_NUMBER() OVER (ORDER BY City) FROM CityTable
```

This example treats all the rows as a single partition. It orders the rows by the City value and returns the following:

Name	City	Window_3
Able	New York	4
Betty	Boston	1
Charlie	Paris	5
Davis	Boston	2
Eve	Paris	6
Francis	Paris	7
George	London	3
Beatrix	Paris	8

```
SELECT Name,City,ROW_NUMBER() OVER (Partition BY City ORDER BY Name) FROM CityTable
```

This example partitions the rows by the City value, orders each City partition by Name values, and returns the following:

Name	City	Window_3
Able	New York	1
Betty	Boston	1
Charlie	Paris	2
Davis	Boston	2
Eve	Paris	3
Francis	Paris	4
George	London	1
Beatrix	Paris	1

NULL

The PARTITION BY clause treats rows with fields that are NULL (have no assigned value) as a partitioned group. For example, `ROW_NUMBER() OVER (Partition BY City)` would assign rows with no City value sequential integers, just as it assigns sequential integers to rows with a City value of 'Paris'.

The ORDER BY clause treats rows with fields that are NULL (have no assigned value) as ordered before any assigned value (having the lowest collation value). For example, `ROW_NUMBER() OVER (ORDER BY City)` would first assign sequential integers to rows with no City value, then assign sequential integers to rows with a City value in collation sequence.

The ROWS clause treats with fields that are NULL (have no assigned value) as having a value of zero. For example, `SUM(Scores) OVER (ORDER BY Scores ROWS 1 PRECEDING)/2` would assign 0.00 to all rows with no Scores value ($(0 + 0) / 2$), and handle the first Scores value by adding 0 to it then dividing by 2.

Supported Window Functions

The following window functions are supported:

- AVG(*field*) — assigns the average of the values in the *field* column for rows within the specified window frame to all rows in that window frame. For example, `AVG(Salary) OVER (PARTITION BY Department) FROM Company.Employee` could be used to compare each employee's salary against the average of the salaries earned by employees within that employee's department. AVG() supports the ROWS clause. Refer to the AVG() function's reference page for further details on use.

- COUNT(* | *field*) — assigns a number to each row in the specified window frame starting at 1. If *field* is specified, the count is only incremented if the contents of *field* is non-null. Otherwise, the count is incremented with every row.

- CUME_DIST() — assigns the cumulative distribution value for all rows within the specified window frame. The cumulative distribution is calculated by counting the rows with values less than or equal to the current row's value and dividing that count by the total number of rows in the window. The column name that the cumulative distribution is computed on is specified in the ORDER BY clause.

- DENSE_RANK() — assigns a ranking integer to each row within the same window frame, starting with 1. The ranking integers are always consecutive, unlike with the RANK() window function. Ranking integers can include duplicates values if multiple rows contain the same value for the window function field.

- FIRST_VALUE(*field*) — assigns the value of the *field* column for the first row (ROW_NUMBER()=1) within the specified window frame to all rows in that window frame. For example, `FIRST_VALUE(Country) OVER (PARTITION BY City)`. FIRST_VALUE() supports the ROWS clause. Note that NULL collates before all values, so if the value of *field* in the first row is NULL, all of the rows in the window will be NULL.

- LAST_VALUE(*field*) — assigns the value of the *field* column for the last row within the specified window frame to all rows in that window frame. LAST_VALUE() supports the ROWS clause.

- LAG(*field*[, *offset*[, *default*]]) — assigns the value of the *field* column for the row that is *offset* rows before the given row within the specified window frame. If no *offset* is specified, the function assigns the value of the *field* column 1 row before the given row by default. By default, LAG() will assign the value NULL if the given row does not have a row *offset* rows before it within its window frame. The user has the option to assign an alternate value under these conditions by including a value *default*.

- LEAD(*field*[, *offset*[, *default*]]) — assigns the value of the *field* column for the row that is *offset* rows after the given row within the specified window frame. If no *offset* is specified, the function assigns the value of the *field* column 1 row after the given row by default. By default, LEAD() will assign the value NULL if the given row does not have a row *offset* rows after it within its window frame. The user has the option to assign an alternate value under these conditions by including a value *default*.

- MAX(*field*) — assigns the maximum value of the *field* column within the specified window frame to all rows in that window frame. For example, `MAX(Salary) OVER (PARTITION BY Department) FROM Company.Employee` could be used to compare each employee's salary against the highest salary earned by an employee within that employee's

department. MAX() supports the ROWS clause. Refer to the MAX() function's reference page for further details on use.

- MIN(*field*) — assigns the minimum value of the *field* column within the specified window frame to all rows in that window frame. For example, `MIN(Salary) OVER (PARTITION BY Department) FROM Company.Employee` could be used to compare each employee's salary against the lowest salary earned by an employee within that employee's department. MIN() supports the ROWS clause. Refer to the MIN() function's reference page for further details on use.

- NTH_VALUE(*field*, *n*) — assigns the value of the *field* column for row number *n* within the specified window frame (counting from 1) to all rows in that window frame. NTH_VALUE() supports the ROWS clause.

- NTILE(*num-groups*) — splits the rows within the specified window frame into a *num-groups* number of groups that each have a roughly equal number of elements. Each group is identified by a number, starting from 1.

- PERCENT_RANK() — assigns a ranking percentage as a fractional number between 0 and 1 (inclusive) to each row within the same window frame. Ranking percentages can include duplicate values if multiple rows contain the same value for the window function field.

- RANK()— assigns a ranking integer to each row within the same window frame, starting with 1. Ranking integers can include duplicate values if multiple rows contain the same value for the window function field.

- ROW_NUMBER() — assigns a unique sequential integer to each row within the same window frame, starting with 1. If multiple rows contain the same value for the window function field, each row is assigned a unique sequential integer.

- SUM(*field*) — assigns the sum of the values of the *field* column within the specified window frame to all rows in that window frame. SUM() supports the ROWS clause. Refer to the SUM() function's reference page for further details on use.

The following example compares the values returned by the ORDER BY clause in these window functions:

```
SELECT Name,City,ROW_NUMBER() OVER (ORDER BY City) AS RowNum,
  RANK() OVER (ORDER BY City) AS RankNum,
  PERCENT_RANK() OVER (ORDER BY City) AS RankPct
  FROM CityTable ORDER BY City
```

This example treats all the rows as a single partition. It orders the rows by the City value and returns the following:

Name	*City*	*RowNum*	*RankNum*	*RankPct*
Harriet		1	1	0
Betty	Boston	2	2	.111111111111111111
Davis	Boston	3	2	.111111111111111111
George	London	4	4	.333333333333333333
Able	New York	5	5	.444444444444444444
Charlie	Paris	6	6	.555555555555555555
Eve	Paris	7	6	.555555555555555555
Francis	Paris	8	6	.555555555555555555
Beatrix	Paris	9	6	.555555555555555555
Jackson	Rome	10	10	1

The ROWS Clause

The ROW clause can be used with the AVG(), FIRST_VALUE(), and SUM() window functions. It can be specified for the other windows functions, but performs no operation (result is the same with or without the ROWS clause).

The ROWS clause has two syntactic forms:

```
ROWS framestart    ROWS BETWEEN framestart AND frameend
```

framestart and *frameend* have five possible values:

```
UNBOUNDED PRECEDING                    /* start at beginning of the current partition */  offset PRECEDING
  /* start offset number of rows preceding the current row */ CURRENT ROW                        /*
start at the current row */  offset FOLLOWING  /* continue offset number of rows following the current
row */  UNBOUNDED FOLLOWING                  /* continue to the end of the current partition */
```

ROWS clause syntax can specify a range in either direction. For example, `ROWS BETWEEN UNBOUNDED PRECEDING AND 1 FOLLOWING` and `ROWS BETWEEN 1 FOLLOWING AND UNBOUNDED PRECEDING` are exactly equivalent.

The ROWS *framestart* syntax defaults to CURRENT ROW as the unspecified second bound of the range. Therefore the following are equivalent:

ROWS UNBOUNDED PRECEDING	ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
ROWS 1 PRECEDING	ROWS BETWEEN 1 PRECEDING AND CURRENT ROW
ROWS CURRENT ROW	ROWS BETWEEN CURRENT ROW AND CURRENT ROW
ROWS 1 FOLLOWING	ROWS BETWEEN CURRENT ROW AND 1 FOLLOWING
ROWS UNBOUNDED FOLLOWING	ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING

The default if the ROWS clause is not specified is ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.

ROWS Clause Example

The following query returns scores that contain a lot of "noise" (random variation). The ROWS clause is used to "smooth" these variations by summing each score with the one immediately preceding it and the one immediately following it in collation sequence, then dividing by 3 to get a rolling average score:

```
SELECT Item,Score,SUM(Score)
  OVER (ORDER BY Score ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)/3
  AS CohortScore FROM Sample.TestScores
```

The operation is: (PrecedingScore + CurrentScore + FollowingScore)/3. Note that the bottom and top CohortScore values will not be accurate, because they are adding 0 to two Score values, then dividing by 3: (0 + CurrentScore + FollowingScore)/3 and (PrecedingScore + CurrentScore + 0)/3.

Using Window Functions

An window function can be used in:

- SELECT list as a listed *select-item*.

   A window function *cannot* be embedded in a subquery or an aggregate function in the *select-item* list.

- ORDER BY clause.

A window function *cannot* be used in and ON, WHERE, GROUP BY, or HAVING clause. Attempting to do so results in an SQLCODE -367 error.

Column Names and Aliases

By default, the column name assigned to the results of a window function is Window_*n*, where the *n* number suffix is the column order number, as specified in the **SELECT** list. Thus, the following example creates column names `Window_3` and `Window_6`:

```
SELECT Name,State,ROW_NUMBER() OVER (PARTITION BY State),Age,AVG(Age),ROW_NUMBER() OVER (ORDER BY Age)
FROM Sample.Person
```

To specify another column name (a column alias), use the AS keyword:

```
SELECT Name,State,ROW_NUMBER() OVER (PARTITION BY State) AS StateRow,Age
FROM Sample.Person
```

You can use a column alias to specify a window field in an ORDER BY clause:

```
SELECT Name,State,ROW_NUMBER() OVER (PARTITION BY State) AS StateRow,Age
FROM Sample.Person
ORDER BY StateRow
```

You cannot use a default column name (such as `Window_3`) in an ORDER BY clause.

For further details on column aliases, refer to the **SELECT** statement.

With ORDER BY

Because an **ORDER BY** clause is applied to the query result set after window functions are evaluated, **ORDER BY** does not affect the values assigned by a *select-item* window function.

See Also

- SELECT statement
- ORDER BY query clause
- Overview of Aggregate Functions

AVG (SQL)

A window function that assigns the average of the values in the *field* column for rows within the specified window frame to all rows in that window frame.

Synopsis

```
AVG(field)
```

Description

AVG returns the average of the values in a specified column for rows within the specified window frame to all rows in that window frame. **AVG** supports the ROWS clause.

This window function works analogously to the aggregate function AVG.

Arguments

field

A column that specifies the rows of values to be averaged.

Examples

The following example compares each employee's salary against the average of the salaries earned by employees within that employee's department:

SQL

```
SELECT AVG(Salary) OVER (PARTITION BY Department) FROM Company.Employee
```

See Also

- Window functions overview

- Aggregate functions: AVG

COUNT (SQL)

A window function that assigns a number to each row in the specified window frame starting at 1.

Synopsis

```
COUNT(*|field)
```

Description

COUNT creates a counter starting at one for each row (or specific rows if the *field* argument is specified).

This window function works analogously to the aggregate function COUNT.

Arguments

field

Specifies which field the count should be incremented in. If *field* is specified, the count is only incremented if the contents of *field* is non-null. Otherwise, the count is incremented with every row.

Examples

The following example counts the number of employees in each department while leaving each employee as their own individual row:

SQL

```
SELECT COUNT(Employee) OVER (PARTITION BY Department) FROM Company.Employee
```

See Also

- Window functions overview

- Aggregate functions: COUNT

CUME_DIST() (SQL)

A window function that assigns the cumulative distribution value for all rows within the specified window frame.

Synopsis

```
CUME_DIST()
```

Description

CUME_DIST() assigns the cumulative distribution for all rows. A cumulative distribution describes the probability of a random variable being at or less than a certain value; this distribution is calculated by counting the rows with values less than or equal to the current row's value and dividing that count by the total number of rows in the window. The column name that the cumulative distribution is computed on is specified in the ORDER BY clause.

Examples

The following example returns the cumulative distribution of salaries for each employee within each department:

SQL

```
SELECT CUME_DIST() OVER (PARTITION BY Department ORDER BY Salary) FROM Company.Employee
```

See Also

- Window functions overview

DENSE_RANK() (SQL)

A window function that assigns a rank to each row within the same window frame, starting with one.

Synopsis

```
DENSE_RANK()
```

Description

DENSE_RANK() assigns a rank (an integer) to each row, where the integers are always consecutive (unlike with the RANK() window function). These ranks are determined by the values specified in the ORDER BY expression; for example, any rows that have the same value are given the same rank. However, unlike **RANK()**, **DENSE_RANK()** does not skip any sequential numbers even if two or more rows share the same rank — if two rows both have a rank of one, the next rank that will be assigned is two.

DENSE_RANK() also allows duplicate values if multiple rows contain the same value for the window function field.

Examples

The following example returns the rank of the employees based on their salaries within each department:

SQL

```
SELECT DENSE_RANK() OVER (PARTITION BY Department ORDER BY Salary) FROM Company.Employee
```

See Also

*   Window functions overview

FIRST_VALUE (SQL)

A window function that assigns the first value of the *field* column within the window frame to each of the other values in that column.

Synopsis

```
FIRST_VALUE(field)
```

Description

FIRST_VALUE uses the value of the *field* column from the first row to assign to all rows in the specific window frame. **FIRST_VALUE** supports the ROWS clause.

Note that the NULL collates before all values, so if the value of *field* in the first row is NULL, all of the rows in the window will be NULL.

Arguments

field

A column that specifies which value to assign to all rows in that window frame.

Examples

The following example returns the first value of the Country column within each city:

SQL

```
SELECT FIRST_VALUE(Country) OVER (PARTITION BY City)
```

See Also

- Window functions overview

LAG (SQL)

A window function that assigns the value of the *field* column for the row that is *offset* rows before the given row within the specified window frame.

Synopsis

```
LAG(field[,offset[,default]])
```

Description

LAG assigns the value of the *field* column for the row that is specified by the *offset*. By default, LAG() will assign the value NULL if the given row does not have a row *offset* rows before it within its window frame.

The user has the option to assign an alternate value under these conditions by including a value *default*.

Arguments

field

A column that specifies the value to be assigned.

offset

This optional argument is an integer specifying from which row to take the value of the *field* column. If no *offset* is specified, the function assigns the value of the *field* column 1 row before the given row by default.

default

This optional argument specifies what value to return if the given row does not have a row *offset* rows before it within its window frame. If no *default* is specified, the value NULL is assigned.

Examples

The following example returns the previous salary for each employee within each department:

SQL

```
SELECT LAG(Salary) OVER (PARTITION BY Department ORDER BY Salary) FROM Company.Employee
```

The following example specifies an *offset* and *default* argument to calculate the salary of the previous employee within each department. If no such employee exists, the value 0 is returned:

SQL

```
SELECT LAG(Salary, 1, 0) OVER (PARTITION BY Department ORDER BY Salary) FROM Company.Employee
```

See Also

- Window functions overview

LAST_VALUE (SQL)

A window function that assigns the last value of the *field* column within the window frame to each of the other values in that column.

Synopsis

```
LAST_VALUE(field)
```

Description

LAST_VALUE uses the value of the *field* column from the last row to assign to all rows in the specific window frame. **LAST_VALUE** supports the ROWS clause.

Note that the NULL collates before all values, so if the value of *field* in the last row is NULL, all of the rows in the window will be NULL.

Arguments

field

A column that specifies which value to assign to all rows in that window frame.

Examples

The following example returns the last value of the Country column within each city:

SQL

```
SELECT LAST_VALUE(Country) OVER (PARTITION BY City)
```

See Also

- Window functions overview

LEAD (SQL)

A window function that assigns the value of the *field* column for the row that is *offset* rows after the given row within the specified window frame.

Synopsis

```
LEAD(field[, offset[, default]])
```

Description

LEAD assigns the value of the *field* column for the row that is specified by the *offset*. By default, LEAD() will assign the value NULL if the given row does not have a row *offset* rows after it within its window frame.

The user has the option to assign an alternate value under these conditions by including a value *default*.

Arguments

field

A column that specifies the value to be assigned

offset

This optional argument is an integer specifying from which row to take the value of the *field* column. If no *offset* is specified, the function assigns the value of the *field* column 1 row after the given row by default.

default

This optional argument specifies what value to return if the given row does not have a row *offset* rows after it within its window frame. If no *default* is specified, the value NULL is assigned.

Examples

The following example returns the subsequent salary for each employee within each department:

SQL

```
SELECT LEAD(Salary) OVER (PARTITION BY Department ORDER BY Salary) FROM Company.Employee
```

The following example specifies an *offset* and *default* argument to calculate the salary of the next employee within each department. If no such employee exists, the value 0 is returned:

SQL

```
SELECT LEAD(Salary, 1, 0) OVER (PARTITION BY Department ORDER BY Salary) FROM Company.Employee
```

See Also

*   Window functions overview

MAX (SQL)

A window function that assigns the maximum value of the *field* column within the specified window frame to all rows in that window frame.

Synopsis

```
MAX(field)
```

Description

MAX assigns the maximum value specified by the *field* column to all rows within the window frame. **MAX** supports the ROWS clause.

Arguments

field

A column that specifies from where to take the maximum value.

Examples

The following example compares each employee's salary against the highest salary earned by an employee within that employee's department:

SQL

```
SELECT MAX(Salary) OVER (PARTITION BY Department) FROM Company.Employee
```

See Also

- Window functions overview

- Aggregate functions: MAX

MIN (SQL)

A window function that assigns the minimum value of the *field* column within the specified window frame to all rows in that window frame.

Synopsis

```
MIN(field)
```

Description

MIN assigns the minimum value specified by the *field* column to all rows within the window frame. **MIN** supports the ROWS clause.

Arguments

field

A column that specifies from where to take the minimum value.

Examples

The following example compares each employee's salary against the lowest salary earned by an employee within that employee's department:

SQL

```
SELECT MIN(Salary) OVER (PARTITION BY Department) FROM Company.Employee
```

See Also

• Window functions overview

• Aggregate functions: MIN

NTH_VALUE (SQL)

A window function that assigns the value of the *field* column for row number *n* within the specified window frame to all rows in the window frame.

Synopsis

```
NTH_VALUE(field, n)
```

Description

NTH_VALUE assigns the value from the *field* column taken from row number *n* to all rows within the window frame. **NTH_VALUE** supports the ROWS clause.

Arguments

field

A column that specifies which value to assign to all rows.

n

A number that denotes which row's *field* column value will be used.

Examples

The following example returns the second highest salary within each department:

SQL

```
SELECT NTH_VALUE(Salary, 2) OVER (PARTITION BY Department ORDER BY Salary) FROM Company.Employee
```

See Also

*   Window functions overview

NTILE (SQL)

A window function that splits the row within the specified window frame into *num-groups* number of groups that each have a roughly equal number of elements.

Synopsis

```
NTILE(num-groups)
```

Description

NTILE splits up the rows in the window frame into *num-groups* groups such that each group contains about the same number of elements. Each group is identified by a number, starting from one.

Arguments

num-groups

A number that specifies how many groups to split the rows into.

Examples

The following example splits up employees within each department into four groups based on their salaries:

SQL

```
SELECT NTILE(4) OVER (PARTITION BY Department ORDER BY Salary) FROM Company.Employee
```

See Also

- Window functions overview

PERCENT_RANK() (SQL)

A window function that assigns a ranking as a fractional number between 0 and 1 (inclusive) to each row within the same window frame.

Synopsis

```
PERCENT_RANK()
```

Description

PERCENT_RANK assigns a percentile ranking between 0 and 1 (inclusive) to each row. Percentile rankings indicate the percentage of data that are at or below a certain value within a group. These ranks can include duplicate values if multiple rows contain the same value for the window function field.

Examples

The following example calculates the percentile rank of each employee based on their salary within each department:

SQL

```
SELECT PERCENT_RANK() OVER (PARTITION BY Department ORDER BY Salary) FROM Company.Employee
```

See Also

- Window functions overview

RANK() (SQL)

A window function that assigns a rank to each row within the same window frame, starting with one.

Synopsis

```
RANK()
```

Description

RANK assigns a rank (an integer) to each row, starting with one. These ranks are determined by the values specified in the ORDER BY expression; for example, any rows that have the same value are given the same rank. However, unlike **DENSE_RANK()**, **RANK()** skips sequential numbers if two or more rows share the same rank — if two rows both have a rank of one, the next rank that will be assigned is three.

The ranks can include duplicate values if multiple rows contain the same value for the window function field.

Examples

The following example assigns a rank to each employee based on their salary within each department:

SQL

```
SELECT RANK() OVER (PARTITION BY Department ORDER BY Salary) FROM Company.Employee
```

See Also

- Window functions overview

ROW_NUMBER() (SQL)

A window function that assigns a unique sequential integer to each row within the same window frame, starting with one.

Synopsis

```
ROW_NUMBER()
```

Description

ROW_NUMBER assigns a unique sequential integer to each row, starting with one. If multiple rows contain the same value for the window function field, each row is assigned a unique sequential integer.

Examples

The following example assigns numbers in sequential order to each employee (within each department) based on their salary:

SQL

```
SELECT ROW_NUMBER() OVER (PARTITION BY Department ORDER BY Salary) FROM Company.Employee
```

See Also

- Window functions overview

SUM (SQL)

A window function that assigns the sum of the values of the *field* column within the specified window frame to all rows in that window frame.

Synopsis

```
SUM(field)
```

Description

SUM adds up the value of the *field* column within the frame to all rows in that frame. **SUM** supports the ROWS clause.

This window function works analogously to the aggregate function SUM.

Arguments

field

A column that specifies the values to be summed up.

Examples

The following example calculates the total salary for each department:

SQL

```
SELECT SUM(Salary) OVER (PARTITION BY Department) FROM Company.Employee
```

See Also

- Window functions overview

- Aggregate functions: SUM

SQL Functions

ABS (SQL)

A numeric function that returns the absolute value of a numeric expression.

Synopsis

```
ABS(numeric-expression)
```

```
{fn ABS(numeric-expression)}
```

Description

ABS returns the absolute value, which is always zero or a positive number. If *numeric-expression* is not a number (for example, the string 'abc', or the empty string '') **ABS** returns 0. **ABS** returns <null> when passed a NULL value.

Note that **ABS** can be used as an ODBC scalar function (with the curly brace syntax) or as an SQL general function.

This function can also be invoked from ObjectScript using the **ABS()** method call:

ObjectScript

```
WRITE $SYSTEM.SQL.Functions.ABS(-0099)
```

Arguments

numeric-expression

A number whose absolute value is to be returned.

ABS returns the same data type as *numeric-expression*.

Examples

The following example shows the two forms of **ABS**:

SQL

```
SELECT ABS(-99) AS AbsGen,{fn ABS(-99)} AS AbsODBC
```

both returns 99.

The following examples show how **ABS** handles some other numbers. InterSystems SQL converts *numeric-expression* to canonical form, deleting leading and trailing zeros and evaluating exponents, before invoking **ABS**.

SQL

```
SELECT ABS(007) AS AbsoluteValue
```

returns 7.

SQL

```
SELECT ABS(-0.000) AS AbsoluteValue
```

returns 0.

SQL

```
SELECT ABS(-99E4) AS AbsoluteValue
```

returns 990000.

SQL

```
SELECT ABS(-99E-4) AS AbsoluteValue
```

returns .0099.

See Also

- SQL functions: CONVERT TO_NUMBER

- ObjectScript function: $ZABS

ACOS (SQL)

A scalar numeric function that returns the arc-cosine, in radians, of a given cosine.

Synopsis

```
{fn ACOS(numeric-expression)}
```

Description

ACOS takes a numeric value and returns the inverse (arc) of its cosine as a floating point number. The value of *numeric-expression* must be a signed decimal number ranging from 1 to -1 (inclusive). A number outside of this range causes a runtime error, generating an SQLCODE -400 (fatal error occurred). **ACOS** returns NULL if passed a NULL value. **ACOS** treats nonnumeric strings, including the empty string (''), as the numeric value 0.

ACOS returns a value with a precision of 19 and a scale of 18.

ACOS can only be used as an ODBC scalar function (with the curly brace syntax).

You can use the DEGREES function to convert radians to degrees. You can use the RADIANS function to convert degrees to radians.

Arguments

numeric-expression

A numeric expression whose value is between -1 and 1. This is the cosine of the angle.

ACOS returns either the NUMERIC or DOUBLE data type. If *numeric-expression* is data type DOUBLE, **ACOS** returns DOUBLE; otherwise, it returns NUMERIC.

Examples

The following examples show the effect of **ACOS** on two cosines:

SQL

```
SELECT {fn ACOS(0.52)} AS ArcCosine
```

returns 1.023945...

SQL

```
SELECT {fn ACOS(-1)} AS ArcCosine
```

returns pi (3.14159...).

See Also

- SQL functions: ASIN, ATAN, COS, COT, SIN, TAN

- ObjectScript function: $ZARCCOS

ASCII (SQL)

A string function that returns the integer ASCII code value of the first (leftmost) character of a string expression.

Synopsis

```
ASCII(string-expression)
```

```
{fn ASCII(string-expression)}
```

Description

ASCII returns NULL if passed a NULL or an empty string value. The returning of NULL for empty string is consistent with SQL Server.

Note that **ASCII** can be invoked as an ODBC scalar function (with the curly brace syntax) or as an SQL general function.

Arguments

string-expression

A string expression, which can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR). A string expression of type CHAR or VARCHAR.

Examples

The following examples both returns 90, which is the ASCII value of the character Z:

SQL

```
SELECT ASCII('Z') AS AsciiCode
```

SQL

```
SELECT {fn ASCII('ZEBRA')} AS AsciiCode
```

InterSystems SQL converts numerics to canonical form before performing **ASCII** conversion. The following example returns 55, which is the ASCII value of the number 7:

SQL

```
SELECT ASCII(+007) AS AsciiCode
```

This number parsing is not done if the numeric is presented as a string. The following example returns 43, which is the ASCII value of the plus (+) character:

SQL

```
SELECT ASCII('+007') AS AsciiCode
```

See Also

- SQL functions: CHAR

- ObjectScript functions: $ASCII $ZLASCII $ZWASCII

ASIN (SQL)

A scalar numeric function that returns the arc-sine, in radians, of the sine of an angle.

Synopsis

```
{fn ASIN(numeric-expression)}
```

Description

ASIN returns the inverse (arc) of the sine of an angle as a floating point number. The value of *numeric-expression* must be a signed decimal number ranging from 1 to -1 (inclusive). A number outside of this range causes a runtime error, generating an SQLCODE -400 (fatal error occurred). **ASIN** returns NULL if passed a NULL value. **ASIN** treats nonnumeric strings, including the empty string ("), as the numeric value 0.

ASIN returns a value with a precision of 19 and a scale of 18.

ASIN can only be used as an ODBC scalar function (with the curly brace syntax).

You can use the DEGREES function to convert radians to degrees. You can use the RADIANS function to convert degrees to radians.

Arguments

numeric-expression

A numeric expression whose value is between -1 and 1. This is the sine of the angle.

ASIN returns either the NUMERIC or DOUBLE data type. If *numeric-expression* is data type DOUBLE, **ASIN** returns DOUBLE; otherwise, it returns NUMERIC.

Examples

The following examples show the effect of **ASIN** on two sines.

SQL

```
SELECT {fn ASIN(0.52)} AS ArcSine
```

returns 0.5468509506...

SQL

```
SELECT {fn ASIN(-1.00)} AS ArcSine
```

returns -1.5707963267...

See Also

- SQL functions: ACOS, ATAN, COS, COT, SIN, TAN
- ObjectScript function: $ZARCSIN

ATAN (SQL)

A scalar numeric function that returns the arc-tangent, in radians, of the tangent of an angle.

Synopsis

```
{fn ATAN(numeric-expression)}
```

Description

ATAN takes any numeric value and returns the inverse (arc) of the tangent of an angle as a floating point number. **ATAN** returns NULL if passed a NULL value. **ATAN** treats nonnumeric strings, including the empty string ("), as the numeric value 0.

ATAN returns a value with a precision of 36 and a scale of 18.

ATAN can only be used as an ODBC scalar function (with the curly brace syntax).

You can use the DEGREES function to convert radians to degrees. You can use the RADIANS function to convert degrees to radians.

Arguments

numeric-expression

A numeric expression. This is the tangent of the angle.

ATAN returns either the NUMERIC or DOUBLE data type. If *numeric-expression* is data type DOUBLE, **ATAN** returns DOUBLE; otherwise, it returns NUMERIC.

Example

The following example shows the effect of **ATAN**:

SQL

```
SELECT {fn ATAN(0.52)} AS ArcTangent
```

returns 0.47951929199...

See Also

- SQL functions: ACOS, ASIN, COS, COT, SIN, TAN

- ObjectScript function: $ZARCTAN

ATAN2 (SQL)

A scalar numeric function that takes two coordinates and returns the arc-tangent angle in radians.

Synopsis

```
{fn ATAN2(y,x)}
```

Description

ATAN2 takes the Cartesian coordinates of a ray (y,x) and returns the inverse (arc) of the tangent of an angle as a floating point number. The signs of both coordinates are used to determine the Cartesian coordinate. When *x* is a positive value, **ATAN2** returns the same value as ATAN(y/x). **ATAN2** returns NULL if passed a NULL value. **ATAN2** treats nonnumeric strings, including the empty string ("), as the numeric value 0.

ATAN2 returns a value with a precision of 36 and a scale of 18.

ATAN2 can only be used as an ODBC scalar function (with the curly brace syntax).

You can use the DEGREES function to convert radians to degrees. You can use the RADIANS function to convert degrees to radians.

Arguments

y

A numeric expression specifying the *y* axis coordinate.

x

A numeric expression specifying the *x* axis coordinate.

ATAN2 returns either the NUMERIC or DOUBLE data type. If *numeric-expression* is data type DOUBLE, **ATAN2** returns DOUBLE; otherwise, it returns NUMERIC.

Example

The following example invokes **ATAN2**:

```
SELECT {fn ATAN2(15,30)} AS ArcTangent
```

returns 0.46

See Also

- SQL functions: ACOS, ASIN, ATAN, COS, COT, SIN, TAN

- ObjectScript function: $ZARCTAN

CAST (SQL)

A function that converts a given expression to a specified data type.

Synopsis

Character Strings

```
CAST(expression AS [ CHAR | CHARACTER | VARCHAR | NCHAR | NVARCHAR ])

CAST(expression AS [ CHAR VARYING | CHARACTER VARYING ])

CAST(expression AS
    [ CHAR(length) | CHARACTER(length) |
    VARCHAR(length) | CHAR VARYING(length) |
    CHARACTER VARYING(length)])
```

Numeric Values

```
CAST(expression AS [INT | INTEGER | BIGINT | SMALLINT | TINYINT])

CAST(expression AS [DEC | DECIMAL | NUMERIC])

CAST(expression AS
    [ DEC(precision,scale) |
    DECIMAL(precision,scale) |
    NUMERIC(precision,scale) ])

CAST(expression AS DOUBLE)

CAST(expression AS [MONEY | SMALLMONEY])
```

Dates and Times

```
CAST(expression AS DATE)

CAST(expression AS TIME)

CAST(expression AS [TIMESTAMP | DATETIME | SMALLDATETIME])

CAST(expression AS POSIXTIME)
```

Bit Values

```
CAST(expression AS BIT)
```

Binary Values

```
CAST(expression AS [BINARY | BINARY VARYING | VARBINARY])

CAST(expression AS [BINARY(length) |
    BINARY VARYING(length) |
    VARBINARY(length)])
```

Unique Identifiers

```
CAST(expression AS GUID)
```

Description

The SQL **CAST** function converts the data type of an expression to the specified data type. For a list of the data types supported by InterSystems SQL, see Data Types.

CAST is similar to **CONVERT**, with these differences:

- **CONVERT** is more flexible than **CAST**. For example, **CONVERT** supports the conversion of stream data and enables formatting of date and time values.

- **CAST** provides more database compatibility than **CONVERT**. Whereas **CAST** is implemented using the ANSI SQL-92 standard, **CONVERT** implementations are database-specific. InterSystems SQL provides **CONVERT** implementations that are compatible with MS SQL Server and ODBC.

If you specify a **CAST** with an unsupported data type, InterSystems IRIS® issues an SQLCODE -376.

Character Strings

- **CAST(***expression* **AS [CHAR | CHARACTER | VARCHAR | NCHAR | NVARCHAR])** converts a numeric or string expression to a character string data type. These data types all map to %Library.String.

    - CHAR, CHARACTER, and NCHAR are equivalent data types and have a default length of 1 character.

    - VARCHAR and NVARCHAR are equivalent data types and have a default length of 30 characters.

  This statement returns `Name` (a character string), `Age` (a numeric value) and `DOB` (a date value) as VARCHAR data types.

  SQL

  ```
  SELECT DISTINCT
    CAST(Name AS VARCHAR) AS VarCharName,
    CAST(Age AS VARCHAR) AS VarCharAge,
    CAST(DOB AS VARCHAR) AS VarCharDOB
  FROM Sample.Person
  ```

  This statement casts a string value to a single character string, truncating the additional letters in the original string.

  SQL

  ```
  SELECT CAST('True' AS CHAR) -- T
  ```

  Example: Cast Character String Values

- **CAST(***expression* **AS [CHAR VARYING | CHARACTER VARYING])** converts the expression and returns the same number of characters in the original value.

  This statement returns the floating-point representation of pi as a string value. The string has the same number of characters as digits in the floating-point precision of pi.

  SQL

  ```
  SELECT CAST({fn PI()} AS CHAR VARYING) AS StringPi -- 3.141592653589793238
  ```

  Example: Cast Character String Values

- **CAST(***expression* **AS [CHAR(***length***) | CHARACTER(***length***) | VARCHAR(***length***) | CHAR VARYING(***length***) | CHARACTER VARYING(***length***)])** converts the expression to a character string with the number of characters specified by *length*. Additional characters are truncated.

  This statement returns a string containing the first 8 characters of the input string expression.

  SQL

  ```
  SELECT CAST('Grabscheid,Alfred N.' AS CHAR(8)) -- Grabsche
  ```

  Example: Cast Character String Values

Numeric Values

- **CAST(***expression* **AS [INT | INTEGER | BIGINT | SMALLINT | TINYINT])** converts the expression to the INT, INTEGER, BIGINT, SMALLINT, or TINYINT data type. In these data types, decimal digits are truncated.

  This example presents an average as an integer, not a floating point. **CAST** truncates the number, so an average age of 42.9 becomes 42.

  SQL

  ```
  SELECT DISTINCT AVG(Age) AS AvgAge,
      CAST(AVG(Age) AS INTEGER) AS IntAvgAge
          FROM Sample.Person
  ```

  Example: Cast Numeric Values

- **CAST(***expression* **AS [DEC | DECIMAL | NUMERIC])** converts the expression to the DEC, DECIMAL, or NUMERIC data types. These data types preserve the number of digits in the original value. InterSystems IRIS converts these data types using $DECIMAL function, which converts $DOUBLE values to $DECIMAL values. These data types map to the %Library.Numeric data type.

  This statement returns the sin of 1 radian, a floating point value, as a decimal.

  SQL

  ```
  SELECT CAST({fn SIN(1)} AS DECIMAL) AS DecimalValue -- 0.841470984807897
  ```

  Example: Cast Numeric Values

- **CAST(***expression* **AS [DEC(***precision*,*scale***) | DECIMAL(***precision*,*scale***) | NUMERIC(***precision*,*scale***)])** specifies the precision and scale of the data type.

  - *precision* specifies the total number of digits that a data type can specify. If specified, *precision* does not affect the value returned by **CAST** but it is retained as part of the defined type.

  - *scale* specifies the total number of decimal digits in the data type. **CAST** rounds numbers to this specified value.

  This statement returns the sin of 1 radian as a decimal value, with four digits after the decimal point.

  SQL

  ```
  SELECT CAST({fn SIN(1)} AS DECIMAL(8,4)) AS ScaledDecimalValue -- 0.8415
  ```

  Example: Cast Numeric Values

- **CAST(***expression* **AS DOUBLE)** converts the expression to the DOUBLE data type, which follows the IEEE floating point standard. For further details, refer to the ObjectScript $DOUBLE function.

  This statement returns the sin of 1 radian as a double value.

  SQL

  ```
  SELECT CAST({fn SIN(1)} AS DOUBLE) AS DoubleValue -- .84147098480789650487
  ```

  Example: Cast Numeric Values

- **CAST(***expression* **AS [MONEY | SMALLMONEY])** converts the expression to a currency numeric data type: MONEY or SMALLMONEY. The scale for currency data types is always 4.

  This statements returns the integer 10 as a currency value.

  ```
  SELECT CAST(10 AS MONEY) AS MoneyValue -- 10.0000 (Display Mode)
  ```

Dates and Times

- **CAST(***expression* **AS DATE)** converts a formatted date expression to the DATE date type. InterSystems IRIS represents dates in these formats, depending on context:

    – The display date format for your locale (for example, mm/dd/yyyy)

    – The ODBC date format (yyyy-mm-dd)

    – The **$HOROLOG** integer date storage format (nnnnn)

  You must specify the **$HOROLOG** date part value as an integer, not a numeric string.

  This statement casts a character string to the DATE data type.

  SQL

  ```
  SELECT CAST('1936-11-26' As DATE) AS StringToDate
  ```

  Example: Cast Formatted Character String to Date

- **CAST(***expression* **AS TIME)** converts a formatted time expression to the TIME data type. InterSystems IRIS represents times in these formats, depending on context:

    – The display time format for your locale (for example, hh:mm:ss)

    – The ODBC date format (hh:mm:ss)

    – The $HOROLOG integer time storage format (nnnnn)

  You must specify the **$HOROLOG** date part value as an integer, not a numeric string.

  This statement casts a character string to the TIME data type.

  SQL

  ```
  SELECT CAST('14:33:45.78' AS TIME) AS StringToTime
  ```

  Example: Cast Formatted Character String to Time

- **CAST(***expression* **AS [TIMESTAMP | DATETIME | SMALLDATETIME])** represents a date and timestamp with the format YYYY-MM-DD hh:mm:ss.nnn. This value corresponds to the ObjectScript $ZTIMESTAMP special variable.

  This statement casts a date and time string to the TIMESTAMP data type.

  SQL

  ```
  SELECT CAST('November 26, 1936 14:33:45.78' AS TIMESTAMP) AS StringToTS
  ```

  Example: Cast Formatted Character String to Timestamp

- **CAST(***expression* **AS POSIXTIME)** converts an expression representing a date and timestamp to an encoded 64-bit signed integer. For more details on this encoding format, see Date, Time, PosixTime, and TimeStamp Data Types.

  This statement casts a date and time string to the POSIXTIME data type.

  ```
  SELECT CAST('November 26, 1936 14:33:45.78' AS POSIXTIME) AS StringToPosix
  ```

  Example: Cast Date to POSIXTIME

Bit Values

- **CAST(*expression* AS BIT)** converts the expression to a single boolean value of data type BIT.

  This statement returns BIT values of 1 and 0, respectively.

  SQL

  ```
  SELECT
    CAST('1' AS BIT) As BitTrue,
    CAST('0' AS BIT) As BitFalse
  ```

  Example: Cast Bit Values

Binary Values

- **CAST(*expression* AS [BINARY | BINARY VARYING | VARBINARY])** converts the expression to one of three data types that map to %Library.Binary (SQLType data type BINARY).

  – BINARY has a default length of 1

  – BINARY VARYING and VARBINARY have a default length of 30.

  When casting to a binary value, **CAST** does not convert the data but it does truncate the length of the value to the specified *length*.

- **CAST(*expression* AS [BINARY(*length*) | BINARY VARYING(*length*) | VARBINARY(*length*)])** sets the maximum character length of the returns binary data type.

Unique Identifiers

- **CAST(*expression* AS GUID)** GUID represents a 36-character value of data type %Library.UniqueIdentifier. If you supply an *expression* longer than 36 characters, **CAST** returns the first 36 characters of *expression*. To generate a GUID value, use the **%SYSTEM.Util.CreateGUID()** method.

Arguments

expression

An SQL expression, commonly a literal or a data field of a table, that is being converted.

length

An integer indicating the maximum number of characters to return after casting.

- If *length* is less than the length of *expression*, the returned data is truncated to the first *length* characters.

- If *length* is greater than the length of *expression*, CAST performs no truncation or padding.

precision

Maximum number of total digits returned in the cast data type, specified as an integer. *precision* is retained as part of the defined data type but does not affect the value returned by **CAST**.

For more details about precision, see Precision and Scale.

scale

Maximum number of decimal digits returned in the cast data type, specified as an integer. CAST rounds the returned value to *scale* number of digits.

- If you specify *scale* = 0, the numeric value is rounded to an integer.

- If you specify *scale* = -1, the numeric value is truncated to an integer.

- If you do not specify *scale*, the scale of the numeric value defaults to 15.

If *scale* is greater than the number of digits in the value being cast, the returned value displays the appropriate number of trailing zeros for Display mode but truncates these digits for Logical and ODBC mode.

For more details about scale, see Precision and Scale.

Examples

Cast Character String Values

You can cast character strings to various numeric, date, time, and character string values.

Cast Between Character Strings

When you cast a character string to another character data type, returning either a single character, the first *length* characters, or the entire character string.

SQL

```
SELECT
  CAST('Hello World' AS CHAR) AS StringToChar, -- H
  CAST('Hello World' AS CHAR(5)) AS StringToCharLength, -- Hello
  CAST('Hello World' AS CHAR VARYING) AS StringToCharVary -- Hello World
```

Before a cast is performed, InterSystems SQL resolves embedded quote characters and string concatenation. Leading and trailing blanks are retained.

SQL

```
SELECT
  CAST('Can''t' AS VARCHAR) AS EmbeddedQuote, -- Can't
  CAST('Can'||'not' AS VARCHAR) AS StringConcatenation -- Cannot
```

Cast Character String to Numeric Type

When you cast a character string to a numeric type, InterSystems SQL returns the single digit zero (0).

```
SELECT CAST('Hello World' AS DOUBLE) AS StringToNumeric -- 0
```

This example shows what happens when you use the **CAST** function to convert Name (a character string) to different numeric data types. In every case, the value returned is 0 (zero).

SQL

```
SELECT DISTINCT
  CAST(Name As INT) As IntName,
  CAST(Name As SMALLINT) As SmallIntName,
  CAST(Name As DEC) As DecName,
  CAST(Name As NUMERIC) As NumericName
FROM Sample.Person
```

Cast Formatted Character String to Date

You can cast strings of the format `'yyyy-mm-dd'` to the DATE data type. This string format corresponds to ODBC date format. InterSystems SQL performs value and range checking on the input expression, where:

- The year (`yyyy`) must be between 00001 and 9999 (inclusive).

- The month (`mm`) must be between 01 and 12 (inclusive).

- The day (dd) must be valid for that month.

InterSystems SQL inserts any missing leading zeros. For example:

SQL

```
SELECT CAST('2022-3-1' AS DATE) AS DateValue -- 03/01/2022 (Display Mode)
```

An invalid date returns 1840-12-31 (logical date 0). For example, 2/29 is valid only on leap days.

SQL

```
SELECT CAST('2021-02-29' AS DATE) AS InvalidDate -- 12/31/1840 (Display Mode)
```

The display mode and the locale's date display format determines the display of the cast. For example, '2004-11-23' might display as '11/23/2004'.

Embedded SQL returns the cast as the corresponding **$HOROLOG** date integer. An invalid ODBC date or a non-numeric string is represented as 0 in logical mode when cast to DATE. Date 0 is displayed as 1840-12-31.

Cast Formatted Character String to Time

You can cast strings of the format 'hh:mm', 'hh:mm:ss' or 'hh:mm:ss.nn', with any number of *length* fractional second digits, to the TIME data type. This string format corresponds to ODBC time format. InterSystems SQL performs value and range checking on the input expression, where:

- The hour (hh) must be from 00 to 23 (inclusive).

- The minute (mm) must be from 00 to 59 (inclusive).

- The day (ss) must be from 00 and to up but not including 60. Fractional seconds are permitted but truncated.

InterSystems SQL adds missing zeros. For example:

SQL

```
SELECT
  CAST('2:45' AS TIME) AS StringToTime1, -- 02:45:00 (Display Mode)
  CAST('2:45:59' AS TIME) AS StringToTime2, -- 02:45:59 (Display Mode)
  CAST('2:45:59.98' AS TIME) AS StringToTime3 -- 02:45:59.98 (Display Mode)
```

An invalid time returns 00:00:00 (logical time 0).

SQL

```
SELECT CAST('11:52:60' AS TIME) AS InvalidTime -- 00:00:00 (Display Mode)
```

Embedded SQL returns the cast as the corresponding **$HOROLOG** time integer. An invalid ODBC time or a non-numeric string is represented as 0 in logical mode when cast to TIME. Time 0 is displayed as 00:00:00.

Cast Formatted Character String to Timestamp

You can cast a string consisting of a valid date and time, a valid date, or a valid time to the TIMESTAMP data type. The date portion can be in a variety of formats, as described in the TO_TIMESTAMP function. The resulting timestamp is in the format: YYYY-MM-DD hh:mm:ss.

SQL

```
SELECT
  CAST('1 MAR 2022 1:33pm' AS TIMESTAMP) As DateToTS1, -- 2022-03-01 13:33:00
  CAST('3/1/2022 13:33:00' AS TIMESTAMP) As DateToTS2 -- 2022-03-01 13:33:00
```

CAST resolves formatted dates as follows:

- Set the date portion (if omitted) to 1841-01-01 (logical date 1).

- Set the time portion (if omitted) to 00:00:00.

- Insert leading zeros (if omitted) for the month and day.

You can precede fractional seconds (if specified) with either a period (.) or a colon (:).

- A period indicates a standard fraction. For example:

  - `12:00:00.4` = four-tenths of a second

  - `12:00:00.004` = four-thousandths of a second

- A colon indicates that what follows is in thousandths of a second. `12:00:00:4` indicates four-thousandths of a second. The permitted number of digits following a colon is limited to three.

You can also cast a character string from one time data type to another. This example casts a character string to the TIME data type, then casts the resulting time to the TIMESTAMP data type. The date is set to the current system date.

SQL

```
SELECT CAST(CAST('14:33:45.78' AS TIME) As TIMESTAMP) AS TimeToTstamp
```

Cast Date Values

You can cast a date to a character string data type, numeric data type, or to another date data type.

Cast Date to Character String

Casting a date to a character data type returns either the complete date or as much of the date as the length of the data type permits. For example, instead of returning the current date as `"yyyy-mm-dd"`, this query returns only `"yyyy-"`.

SQL

```
SELECT CAST(CURRENT_DATE AS CHAR(5)) AS TruncatedDate
```

The CHAR VARYING and CHARACTER VARYING data types return the complete display format.

SQL

```
SELECT CAST(CURRENT_TIME AS CHAR VARYING) AS FullDate
```

If a date displays in a different format, such as `mm/dd/yyyy`, character string data types return the date in ODBC date format (yyyy-mm-dd). For example:

SQL

```
SELECT CAST(TO_DATE('03/01/2022','MM/DD/YYYY') AS VARCHAR) AS DateFormat -- 2022-03-01
```

Cast Date to Numeric Type

Casting a date to a numeric data type returns the **$HOROLOG** value for the date. This is an integer value representing the number of days since Dec. 31, 1840. For example:

SQL

```
SELECT CAST(TO_DATE('01 MAR 2022') AS DECIMAL) AS DateToNumeric -- 66169
```

Cast Date to POSIXTIME

Casting a date to the POSIXTIME data type returns a timestamp as an encoded 64-bit signed integer. Since a date does not have a time portion, the time portion is supplied to the timestamp encoding as 00:00:00.

```
SELECT CAST(CURRENT_DATE AS POSIXTIME) As PosixDate
```

CAST performs date validation. If the *expression* value is not a valid date, it issues an SQLCODE -400 error.

Cast Date to TIMESTAMP, DATETIME, or SMALLDATETIME

Casting a date to the TIMESTAMP, DATETIME, or SMALLDATETIME data type returns a timestamp of the format `YYYY-MM-DD hh:mm:ss`. Since a date does not have a time portion, the time portion of the resulting timestamp is always 00:00:00. **CAST** performs date validation. If the *expression* value is not a valid date, it issues an SQLCODE -400 error.

This example casts a DATE data type column to TIMESTAMP. The POSIXTIME data type is included for comparison.

```
SELECT TOP 5
  DOB,
  CAST(DOB AS TIMESTAMP) AS TStamp,
  CAST(DOB AS POSIXTIME) AS Posix
FROM Sample.Person
```

Cast Numeric Values

You can cast numeric values to a numeric or character data type. When casting a numeric value results in a shortened value, the numeric is truncated, not rounded.

SQL

```
SELECT
  CAST(98.765 AS INT) AS TruncatedInt, -- 98
  CAST(98.765 AS CHAR) AS TruncatedChar1, -- 9
  CAST(98.765 AS CHAR(4)) AS TruncatedChar2 --98.7
```

Casting a negative number to CHAR returns just the negative sign. Casting a fractional number to CHAR returns just the decimal point.

SQL

```
SELECT
  CAST(-50 AS CHAR) AS Negative, -- negative sign: -
  CAST(1/4 AS CHAR) AS Fraction -- decimal point: .
```

A numeric value can contain these values:

- Digits 0 through 9

- A decimal point

- One or more leading signs (+ or –)

- The exponent sign (the letter E or e) followed by, at most, one + or – sign

Before a cast is performed, InterSystems SQL resolves a numeric to its canonical form by performing exponentiation, resolving multiple signs, and stripping the leading plus sign, trailing decimal point, and any leading or trailing zeros. For example:

SQL

```
SELECT
  CAST(1e2 AS DECIMAL(6,2)) AS PositiveExponent, -- 100.000
  CAST(1e-2 AS DECIMAL(6,2)) AS NegativeExponent, -- 0.01
  CAST(+1000 AS DECIMAL(6,2)) AS LeadingSign, -- 1000.00
  CAST(-+1000 AS DECIMAL(6,2)) AS MultipleSigns, -- -1000.00
  CAST(00.100 AS DECIMAL(6,2)) AS LeadingTrailingZeros -- 0.10
```

InterSystems SQL treats double negative signs as a comment indicator. Encountering double negative signs in a number results in InterSystems IRIS processing the remainder of that line of code as a comment. A numeric cannot contain group separator characters (commas). For more details, see Literals.

You can convert numeric values to a variety of numeric types. This example shows how **CAST** converts a floating point number, pi, to different numeric data types. For the integer data types, InterSystems SQL applies truncation.

SQL

```
SELECT
    CAST({fn PI()} As INTEGER) As IntegerPi, -- 3
    CAST({fn PI()} As SMALLINT) As SmallIntPi, -- 3
    CAST({fn PI()} As DECIMAL) As DecimalPi, -- 3.141592653589793
    CAST({fn PI()} As NUMERIC) As NumericPi, -- 3.141592653589793
    CAST({fn PI()} As DOUBLE) As DoublePi -- 3.1415926535897931159
```

In this example, InterSystems IRIS parses the precision and scale values and changes the value returned by **CAST**.

SQL

```
SELECT
    CAST({fn PI()} As DECIMAL) As DecimalPi, -- 3.141592653589793
    CAST({fn PI()} As DECIMAL(6,3)) As DecimalPiPS -- 3.142
```

When a numeric value is cast to a date or time data type, it displays in SQL as zero (0). When a numeric that is cast as a date or time is passed out of embedded SQL to ObjectScript, it displays as the corresponding **$HOROLOG** value.

Cast Bit Values

To return *expression* as a 0 or 1, you can cast it to a BIT value.

CAST returns 1 (true) when *expression* is one of these values:

- The number 1 or any other non-zero numeric value.

- The word "TRUE", "True", "true", or any other combination of uppercase and lowercase letters that spell the word *true*. It cannot be abbreviated to "T".

These **CAST** operations all return 1.

SQL

```
SELECT CAST(1 AS BIT) AS One,
  CAST(7 AS BIT) AS Num,
  CAST(743.6 AS BIT) AS Frac,
  CAST(0.3 AS BIT) AS Zerofrac,
  CAST('tRuE' AS BIT) AS TrueWord
```

CAST returns 0 (false) when *expression* is one of these values:

- Any non-numeric value other than the word true and its various uppercase and lowercase combinations.

- The empty string ('').

- The number 0.

These **CAST** operations all return 0.

SQL

```
SELECT CAST(0 AS BIT) AS Zero,
  CAST('FALSE' AS BIT) AS FalseWord,
  CAST('T' AS BIT) AS T,
  CAST('F' AS BIT) AS F,
  CAST(0.0 AS BIT) AS Zerodot,
  CAST('' AS BIT) AS EmptyString
```

More About

Cast NULL and Empty String Values

Casting NULL to any data type returns NULL.

SQL

```
SELECT CAST(NULL AS DATE) AS NullValue
```

The result of casting an empty string (`''`) depends on the data type.

Data Type	Return Value of Empty String
Character data types	Empty string (`''`)
Numeric data type	0 (zero), with the appropriate number of trailing fractional zeros. The DOUBLE data type returns zero with no trailing fractional zeros.
DATE data type	12/31/1840 (logical date 0)
TIME data type	00:00:00 (logical time 0)
TIMESTAMP, DATETIME, and SMALLDATETIME data types	Empty string (`''`)
BIT data type	0
All binary data types	Empty string (`''`)

See Also

- Data Types

- CONVERT

- TO_CHAR, TO_DATE, TO_NUMBER, TO_POSIXTIME, TO_TIMESTAMP

CEILING (SQL)

A numeric function that returns the smallest integer greater than or equal to a given numeric expression.

Synopsis

```
CEILING(numeric-expression)
```

```
{fn CEILING(numeric-expression)}
```

Description

CEILING returns the nearest integer value greater than or equal to *numeric-expression*. The returned value has a scale of 0. When *numeric-expression* is a NULL value, an empty string (''), or any nonnumeric string, **CEILING** returns NULL.

Note that **CEILING** can be invoked as an ODBC scalar function (with the curly brace syntax) or as an SQL general function.

This function can also be invoked from ObjectScript using the **CEILING()** method call:

```
$SYSTEM.SQL.Functions.CEILING(numeric-expression)
```

Arguments

numeric-expression

A number whose ceiling is to be calculated.

CEILING returns the same data type as *numeric-expression*.

Examples

The following examples show how **CEILING** converts a fraction to its ceiling integer:

SQL

```
SELECT CEILING(167.111) AS CeilingNum1,
       CEILING(167.456) AS CeilingNum2,
       CEILING(167.999) AS CeilingNum3
```

all return 168.

SQL

```
SELECT {fn CEILING(167.00)} AS CeilingNum1,
       {fn CEILING(167.00)} AS CeilingNum2
```

return 167.

SQL

```
SELECT CEILING(-167.111) AS CeilingNum1,
       CEILING(-167.456) AS CeilingNum2,
       CEILING(-167.999) AS CeilingNum3
```

all return -167.

SQL

```
SELECT CEILING(-167.00) AS CeilingNum
```

returns -167.

The following example uses a subquery to reduce a large table of US Zip Codes (postal codes) to one representative city for each ceiling Latitude integer:

SQL

```
SELECT City,State,CEILING(Latitude) AS CeilingLatitude
FROM (SELECT City,State,Latitude,CEILING(Latitude) AS CeilingNum
      FROM Sample.USZipCode)
GROUP BY CeilingNum
ORDER BY CeilingNum DESC
```

See Also

- FLOOR

- ROUND

CHAR (SQL)

A string function that returns the character that has the ASCII code value specified in a string expression.

Synopsis

```
CHAR(code-value)  {fn CHAR(code-value)}
```

Description

CHAR returns the character that corresponds to the specified integer code value. Because InterSystems IRIS is a Unicode system, you can specify the integer code for any Unicode character, 0 through 65535. **CHAR** returns NULL if *code-value* is a integer that exceeds the permissible range of values.

CHAR returns an empty string (") if *code-value* is a nonnumeric string. **CHAR** returns NULL if passed a NULL value.

Note that **CHAR** can be used as an ODBC scalar function (with the curly brace syntax) or as an SQL general function.

Arguments

code-value

An integer code that corresponds to a character.

Examples

The following examples both return the character Z:

SQL

```
SELECT CHAR(90) AS CharCode
```

SQL

```
SELECT {fn CHAR(90)} AS CharCode
```

The following example returns the Greek letter lambda:

SQL

```
SELECT {fn CHAR(955)} AS GreekLetter
```

See Also

- SQL functions: ASCII, CHAR_LENGTH, CHARACTER_LENGTH

- ObjectScript functions: $CHAR, $ZLCHAR, $ZWCHAR

CHARACTER_LENGTH (SQL)

A function that returns the number of characters in an expression.

Synopsis

```
CHARACTER_LENGTH(expression)
```

Description

CHARACTER_LENGTH returns an integer value representing the number of characters, not the number of bytes, in the specified *expression*. The *expression* can be a string, or any other data type such as a numeric or a data stream field. This integer count returned including leading and trailing blanks and the string-termination character. **CHARACTER_LENGTH** returns NULL if passed a NULL value, and 0 if passed an empty string (") value.

Numbers are parsed to canonical form before counting the characters; quoted number strings are not parsed. In the following example, the first **CHARACTER_LENGTH** returns 1 (because number parsing removes leading and trailing zeros), the second **CHARACTER_LENGTH** returns 8.

SQL

```
SELECT CHARACTER_LENGTH(007.0000) AS NumLen,
       CHARACTER_LENGTH('007.0000') AS NumStringLen
```

Note: The **CHARACTER_LENGTH**, **CHAR_LENGTH**, and **DATALENGTH** functions are identical. All of them accept a stream field argument. The **LENGTH** and **$LENGTH** functions do not accept a stream field argument.

LENGTH also differs from these functions by stripping trailing blanks and the string-termination character before counting characters. **$LENGTH** also differs from these functions because it returns 0 if passed a NULL value, and 0 if passed an empty string.

Arguments

expression

An expression, which can be the name of a column, a string literal, or the result of another scalar function. The underlying data type can be a character type (such as CHAR or VARCHAR), a numeric, or a data stream.

CHARACTER_LENGTH returns the INTEGER data type.

Examples

The following example returns the number of characters in the state abbreviation field (Home_State) in the Sample.Employee table. (All U.S. states have a two-letter postal abbreviation):

SQL

```
SELECT DISTINCT CHARACTER_LENGTH(Home_State) AS StateLength
    FROM Sample.Employee
```

The following example returns the names of the employees and the number of characters in each employee name, ordered by ascending number of characters:

SQL

```
SELECT Name,
    CHARACTER_LENGTH(Name) AS NameLength
    FROM Sample.Employee
    ORDER BY NameLength
```

The following examples return the number of characters in a character stream field (Notes) and a binary stream field (Picture) in the Sample.Employee table:

SQL

```
SELECT DISTINCT CHARACTER_LENGTH(Notes) AS NoteLen
     FROM Sample.Employee WHERE Notes IS NOT NULL
```

SQL

```
SELECT DISTINCT CHARACTER_LENGTH(Picture) AS PicLen
     FROM Sample.Employee WHERE Picture IS NOT NULL
```

The following example demonstrates how **CHARACTER_LENGTH** handles Unicode characters. **CHARACTER_LENGTH** counts the number of characters, regardless of their byte length:

SQL

```
SELECT CHARACTER_LENGTH($CHAR(960)_"FACE")
```

returns 5.

See Also

- SQL functions: CHAR, CHAR_LENGTH, DATALENGTH, LENGTH, LEN, $LENGTH

- ObjectScript function: $LENGTH

CHARINDEX (SQL)

A string function that returns the position of a substring within a string, with optional search start point.

Synopsis

```
CHARINDEX(substring,string[,start])
```

Description

CHARINDEX searches a string for a substring. If a match is found, it returns the starting position of the first matching substring, counting from 1. If the substring cannot be found, **CHARINDEX** returns 0.

The empty string is a string value. You can, therefore, use the empty string for either string argument value. The *start* argument treats an empty string value as 0. However, note that the ObjectScript empty string is passed to InterSystems SQL as NULL.

NULL is not a string value in InterSystems SQL. For this reason, specifying NULL for either **CHARINDEX** string argument returns NULL.

CHARINDEX cannot use a %Stream.GlobalCharacter field for either the *string* or *substring* argument. Attempting to do so generates an SQLCODE -37 error. You can use the SUBSTRING function to take a %Stream.GlobalCharacter field and return a %String data type value for use by **CHARINDEX**.

CHARINDEX is case-sensitive. Use one of the case-conversion functions to locate both uppercase and lowercase instances of a letter or character string.

This function provides compatibility with Transact-SQL implementations.

CHARINDEX, POSITION, $FIND, and INSTR

CHARINDEX, **POSITION**, **$FIND**, and **INSTR** all search a string for a specified substring and return an integer position corresponding to the first match. **CHARINDEX**, **POSITION**, and **INSTR** return the integer position of the first character of the matching substring. **$FIND** returns the integer position of the first character after the end of the matching substring. **CHARINDEX**, **$FIND**, and **INSTR** support specifying a starting point for substring search. **INSTR** also supports specifying the substring occurrence from that starting point.

The following example demonstrates these four functions, specifying all optional arguments. Note that the positions of *string* and *substring* differ in these functions:

SQL

```
SELECT POSITION('br' IN 'The broken brown briefcase') AS Position,
       CHARINDEX('br','The broken brown briefcase',6) AS Charindex,
       $FIND('The broken brown briefcase','br',6) AS Find,
       INSTR('The broken brown briefcase','br',6,2) AS Inst
```

For a list of functions that search for a substring, refer to String Manipulation.

Arguments

substring

A substring to match within *string*.

string

A string expression that is the target for the substring search.

start

An optional argument that denotes the starting point for substring search, specified as a positive integer. A character count from the beginning of *string*, counting from 1. To search from the beginning of *string*, omit this argument or specify a *start* of 0 or 1. A negative number, the empty string, NULL, or a nonnumeric value is treated as 0.

CHARINDEX returns the INTEGER data type.

Examples

The following example searches a nucleotide sequence for the first occurrence of the substring TTAGGG. It returns 7, the character position of this substring within the string:

SQL

```
SELECT CHARINDEX('TTAGGG','TTAGTCTTAGGGACATTAGGG')
```

The following example searches for all Name field values that contain the substring 'Fred':

SQL

```
SELECT Name
FROM Sample.Person
WHERE CHARINDEX('Fred',Name)>0
```

The following example uses **SUBSTRING** to allow **CHARINDEX** to search the first 1000 characters of a %Stream.GlobalCharacter field containing DNA nucleotide sequences for the first occurrence of the substring TTAGGG:

SQL

```
SELECT CHARINDEX('TTAGGG',SUBSTRING(DNASeq,1,1000)) FROM Sample.DNASequences
```

The following example matches a substring after the first 10 characters:

SQL

```
SELECT CHARINDEX('Re','Reduce, Reuse, Recycle',10)
```

It returns 16.

The following example specifies a *start* location beyond the length of the string:

SQL

```
SELECT CHARINDEX('Re','Reduce, Reuse, Recycle',99)
```

It returns 0.

The following example shows that **CHARINDEX** handles the empty string ('') just like any other string value:

SQL

```
SELECT CHARINDEX('','Fred Astare'),
       CHARINDEX('A',''),
       CHARINDEX('','')
```

In the above example, the first and second **CHARINDEX** functions return 0 (no match). The third returns 1, because the empty string matches the empty string at position 1.

The following example shows that **CHARINDEX** does not treat NULL as a string value. Specifying NULL for either string always returns NULL:

SQL

```
SELECT CHARINDEX(NULL,'Fred Astare'),
       CHARINDEX('A',NULL),
       CHARINDEX(NULL,NULL)
```

See Also

- $FIND function

- INSTR function

- POSITION function

- String Manipulation

CHAR_LENGTH (SQL)

A function that returns the number of characters in an expression.

Synopsis

```
CHAR_LENGTH(expression)
```

Description

CHAR_LENGTH returns an integer value representing the number of characters, not the number of bytes, in the specified *expression*. The *expression* can be a string, or any other data type such as a numeric or a data stream field. This integer count returned including leading and trailing blanks and the string-termination character. **CHARACTER_LENGTH** returns NULL if passed a NULL value, and 0 if passed an empty string (") value.

Numbers are parsed to canonical form before counting the characters; quoted number strings are not parsed. In the following example, the first **CHAR_LENGTH** returns 1 (because number parsing removes leading and trailing zeros), the second **CHAR_LENGTH** returns 8.

SQL

```
SELECT CHAR_LENGTH(007.0000) AS NumLen,
       CHAR_LENGTH('007.0000') AS NumStringLen
```

Note: The **CHAR_LENGTH**, **CHARACTER_LENGTH**, and **DATALENGTH** functions are identical. All of them accept a stream field argument. The **LENGTH** and **$LENGTH** functions do not accept a stream field argument.

LENGTH also differs from these functions by stripping trailing blanks and the string-termination character before counting characters.

$LENGTH also differs from these functions because it returns 0 if passed a NULL value, and 0 if passed an empty string. **$LENGTH** differs from the other length function by returning data type SMALLINT; all the other length functions return data type INTEGER.

Arguments

expression

An expression, which can be the name of a column, a string literal, or the result of another scalar function. The underlying data type can be a character type (such as CHAR or VARCHAR), a numeric, or a data stream.

CHAR_LENGTH returns the INTEGER data type.

Examples

The following example returns the number of characters in the state abbreviation field (Home_State) in the Sample.Employee table. (All U.S. states have a two-letter postal abbreviation):

SQL

```
SELECT DISTINCT CHAR_LENGTH(Home_State) AS StateLength
    FROM Sample.Employee
```

The following example returns the names of the employees and the number of characters in each employee name, ordered by ascending number of characters:

SQL

```
SELECT Name,
     CHAR_LENGTH(Name) AS NameLength
     FROM Sample.Employee
     ORDER BY NameLength
```

The following examples return the number of characters in a character stream field (Notes) and a binary stream field (Picture) in the Sample.Employee table:

SQL

```
SELECT DISTINCT CHAR_LENGTH(Notes) AS NoteLen
     FROM Sample.Employee WHERE Notes IS NOT NULL
```

SQL

```
SELECT DISTINCT CHAR_LENGTH(Picture) AS PicLen
     FROM Sample.Employee WHERE Picture IS NOT NULL
```

The following Embedded SQL example shows how **CHAR_LENGTH** handles Unicode characters. **CHAR_LENGTH** counts the number of characters, regardless of their byte length:

ObjectScript

```
SET a=$CHAR(960)_"FACE"
WRITE !,a
&sql(SELECT CHAR_LENGTH(:a) INTO :b)
IF SQLCODE'=0 {WRITE !,"Error code ",SQLCODE }
ELSE {WRITE !,"The CHAR length is ",b }
```

returns 5.

See Also

- SQL functions: CHAR, CHARACTER_LENGTH, DATALENGTH, LENGTH, LEN, $LENGTH

- ObjectScript function: $LENGTH

COALESCE (SQL)

A function that returns the value of the first expression that is not NULL.

Synopsis

```
COALESCE(expression,expression [,...])
```

Description

The **COALESCE** function evaluates a list of expressions in left-to-right order and returns the value of the first non-NULL expression. If all expressions evaluate to NULL, NULL is returned.

A string is returned unchanged; leading and trailing blanks are retained. A number is returned in canonical form, with leading and trailing zeros removed.

For further details on NULL handling, refer to NULL and the Empty String.

Data Type of Returned Value

Non-numeric expressions (such as strings or dates) must all be of the same data type, and return a value of that data type. Specifying expressions with incompatible data types results in an SQLCODE -378 error with a Datatype mismatch error message. You can use the CAST function to convert an *expression* to a compatible data type.

Numeric expressions may be of different data types. If you specify numeric expressions with different data types, the data type returned is the *expression* data type most compatible with all of the possible result values, the data type with the highest data type precedence.

A literal value (string, number, or NULL) is treated as data type VARCHAR. If you specify only two expressions, a literal value is compatible with a numeric expression: if the first *expression* is the numeric expression, its data type is returned; if the first *expression* is a literal value, the VARCHAR data type is returned.

Arguments

expression

A series of expressions to be evaluated. Multiple expressions are specified as a comma-separated list. This expression list has a limit of 140 expressions.

NULL Handling Functions Compared

The following table shows the various SQL comparison functions. Each function returns one value if the logical comparison tests True (A same as B) and another value if the logical comparison tests False (A not same as B). These functions allow you to perform NULL logical comparisons. You cannot specify NULL in an actual equality (or non-equality) condition comparison.

SQL Function	Comparison Test	Return Value
COALESCE(ex1,ex2,...)	ex = NULL for each argument	True tests next ex argument. If all ex arguments are True (NULL), returns NULL. False returns ex
IFNULL(ex1,ex2) [two-argument form]	ex1 = NULL	True returns ex2 False returns NULL
IFNULL(ex1,ex2) [three-argument form]	ex1 = NULL	True returns ex2 False returns ex3
{fn IFNULL(ex1,ex2)}	ex1 = NULL	True returns ex2 False returns ex1
ISNULL(ex1,ex2)	ex1 = NULL	True returns ex2 False returns ex1
NVL(ex1,ex2)	ex1 = NULL	True returns ex2 False returns ex1
NULLIF(ex1,ex2)	ex1 = ex2	True returns NULL False returns ex1

Examples

The following example takes a series of values and returns the first (value *d*) that is not NULL. Note that the ObjectScript empty string ("") is translated as NULL in InterSystems SQL:

ObjectScript

```
SELECT COALESCE("","","","firstdata","","nextdata")
```

The following example compares the values of two columns in left-to-right order and returns the value of the first non-NULL column. The FavoriteColors column is NULL for some rows; the Home_State column is never NULL. For **COALESCE** to compare the two, FavoriteColors must be cast as a string:

SQL

```
SELECT TOP 25 Name,FavoriteColors,Home_State,
COALESCE(CAST(FavoriteColors AS VARCHAR),Home_State) AS CoalesceCol
FROM Sample.Person
```

The following Dynamic SQL example compares **COALESCE** to the other NULL-processing functions:

ObjectScript

```
SET myquery = "SELECT TOP 50 %ID,"_
              "IFNULL(FavoriteColors,'blank') AS Ifn2Col,"_
              "IFNULL(FavoriteColors,'blank','value') AS Ifn3Col,"_
              "COALESCE(CAST(FavoriteColors AS VARCHAR),Home_State) AS CoalesceCol,"_
              "ISNULL(FavoriteColors,'blank') AS IsnullCol,"_
              "NULLIF(FavoriteColors,$LISTBUILD('Orange')) AS NullifCol,"_
              "NVL(FavoriteColors,'blank') AS NvlCol"_
              " FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

See Also

- CASE command

- IFNULL function

- ISNULL function

- NULLIF function

- NVL function

CONCAT (SQL)

A scalar string function that returns a character string as a result of concatenating two character expressions.

Synopsis

```
{fn CONCAT(string1,string2)}
```

Description

- **{fn CONCAT(*string1,string2*)}** concatenates two strings and returns a concatenated string. This syntax is equivalent to using the concatenate operator (‖). You can also use the **STRING** function to concatenate two or more expressions into a single string.

  This statement selects the top 5 first names and last names from a table, concatenating the LastName and FirstName columns and separating them by a comma.

  SQL

  ```
  SELECT TOP 5
  FirstName, LastName,
  {fn CONCAT({fn CONCAT(LastName, ',')}, FirstName)} AS FullName
  FROM Sample.Person
  ```

FirstName	LastName	FullName
Quigley	Ulman	Ulman,Quigley
Buzz	Woo	Woo,Buzz
Mario	Mastrolito	Mastrolito,Mario
Julie	Noodleman	Noodleman,Julie
Lawrence	Quincy	Quincy,Lawrence

Example: Concatenate Two Strings

Arguments

string1,string2

The string expressions to be concatenated. The expressions can be the name of a column, a string literal, a numeric, or the result of another scalar function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR).

You can concatenate any combination of numerics or numeric strings; the concatenation result is a numeric string. InterSystems SQL converts numerics to canonical form (exponents are expanded and leading and trailing zeros are removed) before concatenation. Numeric strings are not converted to canonical form before concatenation.

You can concatenate leading or trailing blanks to a string. Concatenating a NULL value to a string results in a NULL

Examples

Concatenate Two Strings

This statement concatenates the Home_State and Home_City columns to create a location value. The concatenation is shown twice, using the **CONCAT** function and the concatenate operator.

SQL

```
SELECT TOP 5
   {fn CONCAT({fn CONCAT(HomeCity,', ')}, HomeState)} AS LocationWithConcatFunction,
   HomeCity||', '||HomeState AS LocationWithConcatOperator
FROM Sample.Person
```

LocationWithConcatFunction	LocationWithConcatOperator
Denver, CO	Denver, CO
Boston, MA	Boston, MA
Albuquerque, NM	Albuquerque, NM
Jacksonville, FL	Jacksonville, FL
Lexington, KY	Lexington, KY

This statement concatenates a string and a NULL, which returns a column of NULLs.

SQL

```
SELECT {fn CONCAT(HomeState,NULL)} AS StrNull
FROM Sample.Person
```

StrNull
NULL
NULL
NULL
NULL
NULL

This statement shows that numbers are converted to canonical form before concatenation. To avoid this, you can specify the number as a string, as shown in the second part of this statement.

SQL

```
SELECT TOP 5
   {fn CONCAT(HomeState,0012.00E2)} AS StrNum,
   {fn CONCAT(HomeState,'0012.00E2')} AS StrStrNum
FROM Sample.Person
```

StrNum	StrStrNum
CO1200	CO0012.00E2
MA1200	MA0012.00E2
NM1200	NM0012.00E2
FL1200	FL0012.00E2
KY1200	KY0012.00E2

The statement shows that trailing blank spaces are retained. When you concatenate a two-letter state field with 10 spaces, the length of each value in the concatenated column is 12.

SQL

```
SELECT TOP 5
HomeState,
CHAR_LENGTH({fn CONCAT(HomeState,'          ')}) AS StrSpace
FROM Sample.Person2
```

HomeState	StrSpace
CO	12
MA	12
NM	12
FL	12
KY	12

See Also

- ASCII
- CHAR
- STRING
- SUBSTRING

CONVERT (SQL)

A function that converts a given expression to a specified data type.

Synopsis

```
CONVERT(type,expression)

CONVERT(type,expression,formatCcode)

{fn CONVERT(expression,type)}
```

Description

The **CONVERT** function converts an expression in one data type to a corresponding value in another data type. **CONVERT** is similar to [CAST](), with these differences:

- **CONVERT** is more flexible than **CAST**. For example, **CONVERT** supports the conversion of stream data and enables formatting of date and time values.

- **CAST** provides more database compatibility than **CONVERT**. Whereas **CAST** is implemented using the ANSI SQL-92 standard, **CONVERT** implementations are database-specific. InterSystems SQL provides **CONVERT** implementations that are compatible with MS SQL Server and ODBC.

MS SQL Server Compatibility

This implementation of **CONVERT** is a general InterSystems IRIS® scalar function that is compatible with MS SQL Server. This function supports the formatting of dates and times and the conversion of [stream data]().

- **CONVERT(*type*,*expression*)** converts an expression to the specified data type. For a list of the data types supported by InterSystems SQL, see [Data Types]().

  This statement converts a decimal number (an approximation of pi) to a character string, truncating the number to four characters.

  SQL

  ```
  SELECT CONVERT(CHAR(4),3.14159) -- '3.14'
  ```

  Examples:

  - [Convert Between Numeric Types]()

  - [Convert Between Character Strings]()

  - [Convert Stream Data to Character String]()

  - [Convert Character String to Numeric Type]()

  - [Convert Date to Timestamp]()

  - [Convert Date to Numeric Type]()

- **CONVERT(*type*,*expression*,*formatCode*)** converts the expression to the specified data type and formats the returned value based on the specified format code.

  This statement converts a date string to the TIMESTAMP data type. The function converts the input based on format code 103, which represents the `mm/dd/yy` format. For a complete list of format codes, see the *[formatCode]()* argument.

SQL

```
SELECT CONVERT(TIMESTAMP,'1/1/99',103) -- '01/01/1999 00:00:00'
```

Example: Convert Date to Character String

ODBC Compatibility

This implementation of **CONVERT** is a general InterSystems IRIS ODBC scalar function. This function does not support the formatting of dates and times. It also does not support the conversion of stream data.

- **{fn CONVERT(*expression*,*type*)}** converts the expression to the specified data type. In this implementation of **CONVERT**, you must precede each data type argument with the SQL_ keyword. These data types do not accept parameter. For example, for string data types, you cannot set a maximum length. For numeric data types, you cannot set the precision (maximum number of digits) and scale (maximum number of decimal digits).

  This statement converts a decimal number to a character string. The returned string performs no truncation. Maximum length specifications such as SQL_VARCHAR(4) are not permitted.

  SQL

  ```
  SELECT {fn CONVERT(3.14159,SQL_VARCHAR) } -- '3.14159'
  ```

Examples:

- Convert Between Numeric Types

- Convert Between Character Strings

- Convert Character String to Numeric Type

- Convert Date to Timestamp

- Convert Date to Numeric Type

Arguments

type

The data type to convert *expression* to. The types you can specify depend on whether you are using the InterSystems IRIS **CONVERT()** syntax or the ODBC **{fn CONVERT()}** syntax.

CONVERT() Function

The InterSystems IRIS **CONVERT()** syntax supports the data types described in Data Types. Common data types that you can specify include:

- Character string data types: CHAR, CHARACTER, VARCHAR. For some character string types, you can optionally specify a maximum length parameter. For example: VARCHAR(10)

- Numeric data types: INTEGER, DECIMAL, DOUBLE, MONEY. For some numeric types, you can optionally specify precision and scale parameters. For example: DECIMAL(8,4)

- Data and time data types: DATE, TIME, TIMESTAMP, POSIXTIME

- Bit and binary data types: BIT, BINARY, VARBINARY

{fn CONVERT()} Function

The ODBC **{fn CONVERT()}** syntax supports a more limited set of data types than the **CONVERT()** syntax. The supported data types correspond to the ones you specify for the **CONVERT()** syntax but must be preceded by the SQL_ keyword.

This table describes the valid data types that you can specify, separated into two groups:

- The first group converts both the data value and the data type. For example, converting a %Date source to SQL_VARCHAR transforms the date to a text value and the query processes it as a VARCHAR data type.

- The second group converts the data type but does not convert the data value. For example, converting a %Date source to INTEGER does not transform the %Date source but the query processes the integer form of the date as an INTEGER data type.

Source	Valid Conversion Types (Type and Value Converted)	Valid Conversion Types (Only Type Converted)
Any numeric data type	SQL_VARCHAR, SQL_DOUBLE, SQL_DATE, SQL_TIME	n/a
%String	SQL_DATE, SQL_TIME, SQL_TIMESTAMP	n/a
%Date	SQL_VARCHAR, SQL_POSIXTIME, SQL_TIMESTAMP	SQL_INTEGER, SQL_BIGINT, SQL_SMALLINT, SQL_TINYINT, SQL_DATE
%Time	SQL_VARCHAR, SQL_POSIXTIME, SQL_TIMESTAMP	SQL_INTEGER, SQL_BIGINT, SQL_SMALLINT, SQL_TINYINT, SQL_TIME
%PosixTime	SQL_TIMESTAMP, SQL_DATE, SQL_TIME	SQL_VARCHAR, SQL_INTEGER, SQL_BIGINT, SQL_SMALLINT, SQL_TINYINT
%TimeStamp	SQL_POSIXTIME, SQL_DATE, SQL_TIME	SQL_VARCHAR, SQL_INTEGER, SQL_BIGINT, SQL_SMALLINT, SQL_TINYINT
Any non-stream data type	SQL_INTEGER, SQL_BIGINT, SQL_SMALLINT, SQL_TINYINT	SQL_DOUBLE

When specifying data types for this, keep these points in mind:

- SQL_VARCHAR is the standard ODBC representation. When converting to SQL_VARCHAR, dates and times are converted to their appropriate ODBC representations; numeric datatype values are converted to a string representation. When converting from SQL_VARCHAR, the value must be a valid ODBC Time, Timestamp, or Date representation.

- When converting a time value to SQL_TIMESTAMP or SQL_POSIXTIME, an unspecified date defaults to 1841-01-01. In the **CONVERT**() syntax, the date defaults to 1900-01-01.

- When converting a date value to SQL_TIMESTAMP or SQL_POSIXTIME the time defaults to 00:00:00.

- Fractional seconds can be preceded by either a period (.) or a colon (:). The symbols have different meanings. A period indicates a standard fraction; thus `12:00:00.4` indicates four-tenths of a second, and `12:00:00.004` indicates four-thousandth of a second. A colon indicates that what follows is in thousandths of a second; thus `12:00:00:4` indicates four-thousandth of a second. The permitted number of digits following a colon is limited to three.

- When converting to an integer data type or the SQL_DOUBLE data type, the **CONVERT** function converts data values (including dates and times) to a numeric representation. For SQL_DATE, this is the number of days since January 1, 1841. For SQL_TIME, this is the number of seconds since midnight. When **CONVERT** encounters a nonnumeric character, it truncates the input string at that character. The integer data types also truncate decimal digits, returning the integer portion of the number.

expression

The expression to be converted to a new data type. *expression* can be scalar, such as a single string value, or nonscalar, such as a table column. The valid values of *expression* depend on the data type specified by *type*.

- If *expression* does not have a defined data type (for example, a host variable supplied by ObjectScript) its data type defaults to the string data type.

- If *expression* contains stream data and you are using the **{fn CONVERT(*expression*,*type*)}** syntax, then CONVERT issues an SQLCODE -37 error.

- If *expression* is NULL, the converted value remains NULL, regardless of the specified type.

- If *expression* is an empty string (") or a nonnumeric string value, the returned value depends on the specified *type*:

  – If *type* is a string data type, then **CONVERT** returns the supplied value.

  – If *type* is a numeric data type or type TIME, SQL_TIME, or SQL_DATE, then CONVERT returns 0 (zero).

Specifying an invalid value given the *type* results in an SQLCODE -141 error.

formatCode

An integer code that specifies date, datetime, and time formats.

Use *formatCode* to define the output when converting from a date/time/timestamp data type to a character string. For example:

SQL

```
SELECT CONVERT(VARCHAR,TO_DATE('22 FEB 2022'),1) -- '02/22/22'
```

You can also use *formatCode* to define the input when converting from a character string to a date/time/timestamp data type. For example:

SQL

```
SELECT CONVERT(DATE,'22 FEB 2022',106) -- '02/22/2022'
```

Only the **CONVERT()** syntax supports *formatCode*.

Specifying an *expression* with an invalid format or a format that does not match the *formatCode* generates an SQLCODE -141 error. Specifying a non-existent *formatCode* returns 1900-01-01 00:00:00.

This table describes the supported format codes, where:

- The first column lists codes that output a two-digit year.

- The second column lists code that output a four-digit year or do not output a year.

Two-Digit Year Codes	Four-Digit Year Codes	Format
n/a	0 or 100	Mon dd yyyy hh:mmAM (or PM)
1	101	mm/dd/yy
2	102	yy.mm.dd
3	103	dd/mm/yy
4	104	dd.mm.yy

Two-Digit Year Codes	Four-Digit Year Codes	Format
5	105	dd-mm-yy
6	106	dd Mon yy
7	107	Mon dd, yy (no leading zero when dd < 10)
n/a	8 or 108	hh:mm:ss
n/a	9 or 109	Mon dd yyyy hh:mm:ss:nnnAM (or PM)
10	110	mm-dd-yy
11	111	yy/mm/dd
12	112	yymmdd
n/a	13 or 113	dd Mon yyyy hh:mm:ss:nnn (24 hour)
n/a	14 or 114	hh:mm:ss.nnn (24 hour)
n/a	20 or 120	yyyy-mm-dd hh:mm:ss (24 hour)
n/a	21 or 121	yyyy-mm-dd hh:mm:ss.nnn (24 hour)
n/a	126	yyyy-mm-ddThh:mm:ss.nnn (24 hour)
n/a	130	dd Mon yyyy hh:mm:ss:nnnAM (or PM)
n/a	131	dd/mm/yyyy hh:mm:ss:nnnAM (or PM)

Range of Values

The range of permitted dates is 0001-01-01 through 9999-12-31.

Default Values

For the **CONVERT()** syntax, when converting a time value to TIMESTAMP, POSIXTIME, DATETIME, or SMALL-DATETIME, the date defaults to 1900-01-01. For the **{fn CONVERT()}** syntax, the date defaults to 1841-01-01.

When converting a date value to TIMESTAMP, POSIXTIME, DATETIME, or SMALLDATETIME, the time defaults to 00:00:00.

Default Format

If you do not specify *formatCode*, **CONVERT** tries to determine the format from the specified value. If it cannot, it defaults to *formatCode* 100 (mm-dd-yy).

Two-Digit Years

Two-digit years from 00 through 49 are converted to 21st century dates (2000 through 2049).

Two-digit years from 50 through 99 are converted to 20th century dates (1950 through 1999).

Fractional Seconds

You can precede fractional seconds by either a period (.) or a colon (:). The symbols have different meanings:

- Period (default) — Valid for all *formatCode* values. A period indicates a standard fraction. For example, 12:00:00.4 indicates four-tenths of a second and 12:00:00.004 indicates four-thousandth of a second. **CONVERT** has no limit on the number of digits of fractional precision.

- Colon — Valid only for *formatCode* values 9/109, 13/113, 14/114, 130, and 131. A colon indicates that the number that follows is in thousandths of a second. For example, `12:00:00:4` indicates four-thousandth of a second (`12:00:00.004`). You can specify a maximum of three digits of fractional precision.

Examples

Convert Between Numeric Types

This example compares the conversion of a fractional number using the DECIMAL and DOUBLE data types. It uses the InterSystems IRIS **CONVERT()** syntax. The conversion to DOUBLE results in a loss of precision.

SQL

```
SELECT CONVERT(DECIMAL,-123456789.0000123456789) AS DecimalVal, -- -123456789.0000123457
       CONVERT(DOUBLE,-123456789.0000123456789) AS DoubleVal -- -123456789.00001235306
```

This statement uses the ODBC **{fn CONVERT()}** syntax to perform a similar conversion. This syntax does not support a DECIMAL data type, so the statement converts to a DOUBLE data type only.

SQL

```
SELECT {fn CONVERT(-123456789.0000123456789,SQL_DOUBLE) } AS DecimalVal -- -123456789.00001235306
```

Convert Between Character Strings

This example shows how to truncate a string by performing a VARCHAR-to-VARCHAR conversion, specifying an output string length shorter than the *expression* string length. Truncation is supported only for the InterSystems IRIS **CONVERT()** syntax. The only supported character string format for the ODBC **{fn CONVERT()}** syntax is SQL_VARCHAR.

SQL

```
SELECT CONVERT(VARCHAR(5),'Hello, World') As TruncatedValue -- 'Hello'
```

If a character data type has no specified length, the default maximum length is 30 characters.

SQL

```
SELECT CONVERT(VARCHAR,'This string is more than 30 characters.') --This string is more than 30 ch
```

SQL

```
SELECT {fn CONVERT('This string is more than 30 characters.',SQL_VARCHAR) } --This string is more than
 30 ch
```

For the **CONVERT()** syntax, this maximum length also applies to converts to BINARY or VARBINARY types. Otherwise, these data types with no specified length are mapped to a MAXLEN of 1 character, as shown in the Data Types table.

Convert Stream Data to Character String

This example converts a character stream field to a VARCHAR text string. It also displays the length of the character stream field using CHAR_LENGTH:

SQL

```
SELECT Notes,CONVERT(VARCHAR(80),Notes) AS NoteText,CHAR_LENGTH(Notes) AS TextLen
FROM Sample.Employee WHERE Notes IS NOT NULL
```

The ODBC **{fn CONVERT()}** syntax does not support character streams.

Convert Date to Character String

This example converts dates in the "DOB" (Date Of Birth) column to the SQL_VARCHAR data type. The resulting string is in the format: `yyyy-mm-dd`.

SQL

```
SELECT DOB,CONVERT(VARCHAR,DOB) AS DOBtoVChar
FROM Sample.Person
```

SQL

```
SELECT DOB,{fn CONVERT(DOB,SQL_VARCHAR)} AS DOBtoVChar
FROM Sample.Person
```

This example shows several conversions of the date-of-birth field (DOB) to a formatted character string. A sample output date string appears in a comment after each conversion.

SQL

```
SELECT DOB,
       CONVERT(VARCHAR(20),DOB) AS DOBDefault, -- Mar 20 1983 12:00AM
       CONVERT(VARCHAR(20),DOB,100) AS DOB100, -- Mar 20 1983 12:00AM
       CONVERT(VARCHAR(20),DOB,107) AS DOB107, -- Mar 20, 1983
       CONVERT(VARCHAR(20),DOB,114) AS DOB114, -- 00:00:00.000
       CONVERT(VARCHAR(20),DOB,126) AS DOB126  -- 1983-03-20T00:00:00:
FROM Sample.Person
```

The default format and the code-100 format are the same. Because the DOB field does not contain a time value, formats that display time (here including the default, 100, 114, and 126) supply a zero value, which represents 12:00AM (midnight). The code-126 format provides a date and time string that contains no spaces.

Only the InterSystems IRIS **CONVERT()** syntax supports date string formatting, not ODBC **{fn CONVERT()}** syntax.

Convert Character String to Numeric Type

This example converts a mixed string to an integer. InterSystems IRIS truncates the string at the first nonnumeric character and then converts the resulting numeric to canonical form:

SQL

```
SELECT CONVERT(INTEGER,'007 James Bond') -- 7
```

SQL

```
SELECT {fn CONVERT('007 James Bond',SQL_INTEGER)} -- 7
```

Convert Date to Timestamp

This example converts dates in the "DOB" (Date Of Birth) column to timestamp data types. The resulting timestamp is in the format `yyyy-mm-dd hh:mm:ss`.

SQL

```
SELECT DOB,CONVERT(TIMESTAMP,DOB) AS DOBtoTstamp
FROM Sample.Person
```

SQL

```
SELECT DOB,{fn CONVERT(DOB,SQL_TIMESTAMP)} AS DOBtoTstamp
FROM Sample.Person
```

Convert Date to Numeric Type

This example converts dates in the "DOB" (Date Of Birth) column to integer data types. The resulting integer is the $HOROLOG count of days since December 31, 1840.

SQL

```
SELECT DOB,CONVERT(INTEGER,DOB) AS DOBtoInt
FROM Sample.Person
```

SQL

```
SELECT DOB,{fn CONVERT(DOB,SQL_INTEGER)} AS DOBtoInt
FROM Sample.Person
```

Convert Bit Values

You can perform a BIT data type conversion. The permitted values are 1, 0, or NULL. If you specify any other value, InterSystems IRIS issues an SQLCODE -141 error. This example shows two BIT conversions of a NULL:

ObjectScript

```
SET a=""
&sql(SELECT CONVERT(BIT,:a),
     CONVERT(BIT,NULL)
     INTO :x,:y)
WRITE !,"SQLCODE=",SQLCODE
WRITE !,"the host variable is:",x
WRITE !,"the NULL keyword is:",y
```

You can specify empty strings in bit conversion only when it is stored in a host variable using embedded SQL. If you specify an empty string directly, as in CONVERT(BIT,''), InterSystems IRIS issues an SQLCODE -141 error.

More About

CONVERT Class Method

You can also perform data type conversions using the **CONVERT()** method call, specifying data types as SQL_ keywords, as shown in this syntax:

```
$SYSTEM.SQL.Functions.CONVERT(expression,SQL_convertToType,SQL_convertFromType)
```

For example:

ObjectScript

```
write $SYSTEM.SQL.Functions.CONVERT(66225,"SQL_VARCHAR","SQL_DATE")
```

See Also

- CAST

- Data Types

COS (SQL)

A scalar numeric function that returns the cosine, in radians, of an angle.

Synopsis

```
{fn COS(numeric-expression)}
```

Description

COS takes any numeric value and returns the cosine as a floating point number. The returned value is within the range -1 to 1, inclusive. **COS** returns NULL if passed a NULL value. **COS** treats nonnumeric strings as the numeric value 0.

COS returns a value with a precision of 19 and a scale of 18.

COS can only be used as an ODBC scalar function (with the curly brace syntax).

You can use the DEGREES function to convert radians to degrees. You can use the RADIANS function to convert degrees to radians.

Arguments

numeric-expression

A numeric expression. This is an angle expressed in radians.

COS returns either the NUMERIC or DOUBLE data type. If *numeric-expression* is data type DOUBLE, **COS** returns DOUBLE; otherwise, it returns NUMERIC.

Examples

These examples show the effect of **COS** on two sines.

SQL

```
SELECT {fn COS(0.52)} AS Cosine
```

returns 0.86781.

SQL

```
SELECT {fn COS(-.31)} AS Cosine
```

returns 0.95233.

See Also

*   SQL functions: ACOS, ASIN, ATAN, COT, SIN, TAN

*   ObjectScript function: $ZCOS

COT (SQL)

A scalar numeric function that returns the cotangent, in radians, of an angle.

Synopsis

```
{fn COT(numeric-expression)}
```

Description

COT takes any nonzero number and returns its cotangent as a floating point number. **COT** returns NULL if passed a NULL value. A numeric value of 0 (zero) causes a runtime error, generating an SQLCODE -400 (fatal error occurred). **COT** treats nonnumeric strings as the numeric value 0.

COT returns a value with a precision of 36 and a scale of 18.

COT can only be used as an ODBC scalar function (with the curly brace syntax).

You can use the DEGREES function to convert radians to degrees. You can use the RADIANS function to convert degrees to radians.

Arguments

numeric-expression

A numeric expression. This is an angle expressed in radians.

COT returns either the NUMERIC or DOUBLE data type. If *numeric-expression* is data type DOUBLE, **COT** returns DOUBLE; otherwise, it returns NUMERIC.

Examples

The following examples show the effect of **COT**:

SQL

```
SELECT {fn COT(0.52)} AS Cotangent
```

returns 1.74653.

SQL

```
SELECT {fn COT(124.1332)} AS Cotangent
```

returns -0.040312.

See Also

- SQL functions: ACOS, ASIN, ATAN, COS, SIN, TAN

- ObjectScript function: $ZCOT

CURDATE (SQL)

A scalar date/time function that returns the current local date.

Synopsis

```
{fn CURDATE()}
{fn CURDATE}
```

Description

CURDATE takes no arguments. It returns the current local date as data type DATE. Note that the argument parentheses are optional. **CURDATE** returns the current local date for this timezone; it adjusts for local time variants, such as Daylight Saving Time.

CURDATE in Logical mode returns the current local date in $HOROLOG format; for example, 64701. **CURDATE** in Display mode returns the current local date in the default format for the locale. For example, in an American locale 02/22/2018, in a European locale 22/02/2018, in a Russian locale 22.02.2018.

To specify a different date format, use the TO_DATE function. To change the default date format, use the SET OPTION command with the DATE_FORMAT, YEAR_OPTION, or DATE_SEPARATOR options.

To return just the current date, use **CURDATE** or CURRENT_DATE. These functions return their values in DATE data type. The CURRENT_TIMESTAMP, GETDATE and NOW functions can also be used to return the current date and time as a TIMESTAMP data type.

Note that all InterSystems SQL time and date functions except GETUTCDATE are specific to the local time zone setting. To get a current timestamp that is universal (independent of time zone) you can use **GETUTCDATE** or the ObjectScript $ZTIMESTAMP special variable.

These data types perform differently when using embedded SQL. The DATE data type stores values as integers in $HOROLOG format; when displayed in SQL they are converted to date display format; when returned from embedded SQL they are returned as integers. A TIMESTAMP data type stores and displays its value in the same format. You can use the CONVERT function to change the data type of dates and times.

Examples

The following examples both return the current date:

SQL

```
SELECT {fn CURDATE()} AS Today
```

SQL

```
SELECT {fn CURDATE} AS Today
```

The following example returns the current date. Because this date is stored in $HOROLOG format, it is returned as an integer:

SQL

```
SELECT {fn CURDATE()} AS CurrentDate
```

The following example shows how **CURDATE** can be used in a **SELECT** statement to return all records that have a shipment date that is the same or later than today's date:

SQL

```
SELECT * FROM Orders
    WHERE ShipDate >= {fn CURDATE()}
```

See Also

- SQL functions: CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, CURTIME, GETDATE, GETUTCDATE, NOW

- ObjectScript function: $ZDATE

CURRENT_DATE (SQL)

A date/time function that returns the current local date.

Synopsis

```
CURRENT_DATE
```

Description

CURRENT_DATE takes no arguments. It returns the current local date as data type DATE. Argument parentheses are not permitted. **CURRENT_DATE** returns the current local date for this timezone; it adjusts for local time variants, such as Daylight Saving Time.

CURRENT_DATE in Logical mode returns the current local date in $HOROLOG format; for example, 64701. **CURRENT_DATE** in Display mode returns the current local date in the default format for the locale. For example, in an American locale 02/22/2018, in a European locale 22/02/2018, in a Russian locale 22.02.2018.

To specify a different date format, use the TO_DATE function. To change the default date format, use the SET OPTION command with the DATE_FORMAT, YEAR_OPTION, or DATE_SEPARATOR options.

To return just the current date, use **CURRENT_DATE** or CURDATE. These functions return their values in DATE data type. The CURRENT_TIMESTAMP, GETDATE and NOW functions can also be used to return the current date and time as a TIMESTAMP data type.

Note that all InterSystems SQL time and date functions except GETUTCDATE are specific to the local time zone setting. To get a current timestamp that is universal (independent of time zone) you can use **GETUTCDATE** or the ObjectScript $ZTIMESTAMP special variable.

These data types perform differently when using embedded SQL. The DATE data type stores values as integers in $HOROLOG format; when displayed in SQL they are converted to date display format; when returned from embedded SQL they are returned as integers. A TIMESTAMP data type stores and displays its value in the same format. You can use the CONVERT function to change the datatype of dates and times.

CURRENT_DATE can be used as a default specification keyword in CREATE TABLE or ALTER TABLE.

Examples

The following example returns the current date, converted to Display mode:

SQL

```
SELECT CURRENT_DATE AS Today
```

The following example also returns the current date, but because this date is stored in $HOROLOG format, it is returned as an integer:

SQL

```
SELECT CURRENT_DATE
```

The following example shows how **CURRENT_DATE** can be used in a **WHERE** clause to return records of people born in the last 1000 days:

SQL

```
SELECT Name, DOB, Age
FROM Sample.Person
WHERE DOB > CURRENT_DATE - 1000
```

See Also

CURDATE, CURRENT_TIME, CURRENT_TIMESTAMP, CURTIME, GETDATE, GETUTCDATE, NOW

CURRENT_TIME (SQL)

A date/time function that returns the current local time.

Synopsis

```
CURRENT_TIME CURRENT_TIME(precision)
```

Description

CURRENT_TIME returns the current local time for this timezone. It adjusts for local time variants, such as Daylight Saving Time.

CURRENT_TIME takes either no arguments or a precision argument. Empty argument parentheses are not permitted.

CURRENT_TIME in Logical mode returns the current local time in $HOROLOG format; for example, 37065. **CURRENT_TIME** in Display mode returns the current local time in the default format for the locale; for example, 10:18:27.

To change the default time format, use the SET OPTION command with the TIME_FORMAT and TIME_PRECISION options. You can configure fractional seconds of precision, as described below.

To return just the current time, use **CURRENT_TIME** or CURTIME. These functions return their values in TIME data type. The CURRENT_TIMESTAMP, GETDATE and NOW functions can also be used to return the current date and time as a TIMESTAMP data type.

Note that all InterSystems SQL time and date functions except GETUTCDATE are specific to the local time zone setting. To get a current timestamp that is universal (independent of time zone) you can use **GETUTCDATE** or the ObjectScript $ZTIMESTAMP special variable.

These data types perform differently when using embedded SQL. The TIME data type stores values as integers in $HOROLOG format (as the number of seconds since midnight); when displayed in SQL they are converted to time display format; when returned from embedded SQL they are returned as integers. A TIMESTAMP data type stores and displays its value in the same format. You can use the CAST or CONVERT function to change the datatype of times and dates.

CURRENT_TIME can be used as a default specification keyword in CREATE TABLE or ALTER TABLE. **CURRENT_TIME** cannot specify a *precision* argument when used as a default specification keyword.

Fractional Seconds Precision

CURRENT_TIME can return up to nine digits of fractional seconds of precision. The default for the number of digits of precision can be configured using the following:

- SET OPTION with the TIME_PRECISION option.

- The system-wide **$SYSTEM.SQL.Util.SetOption**() method configuration option DefaultTimePrecision. To determine the current setting, call **$SYSTEM.SQL.CurrentSettings**() which displays Default time precision; the default is 0.

- Go to the Management Portal, select **System Administration**, **Configuration**, **SQL and Object Settings**, **SQL**. View and edit the current setting of **Default time precision for GETDATE(), CURRENT_TIME, and CURRENT_TIMESTAMP**.

Specify an integer 0 through 9 (inclusive) for the default number of decimal digits of precision to return. The default is 0. The actual precision returned is platform dependent; digits of precision in excess of the precision available on your system are returned as zeroes.

Arguments

precision

A positive integer that specifies the time precision as the number of digits of fractional seconds. The default is 0 (no fractional seconds); this default is configurable.

CURRENT_TIME returns the TIME data type.

Examples

The following example returns the current system time:

SQL

```
SELECT CURRENT_TIME
```

The following example returns the current system time with three digits of fractional seconds precision:

SQL

```
SELECT CURRENT_TIME(3)
```

The following Embedded SQL example returns the current time. Because this time is stored in $HOROLOG format, it is returned as an integer:

ObjectScript

```
&sql(SELECT CURRENT_TIME INTO :a)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"Current time is: ",a }
```

The following example sets the LastCall field in the selected row of the Contacts table to the current system time:

SQL

```
UPDATE Contacts SET LastCall = CURRENT_TIME
  WHERE Contacts.ItemNumber=:item
```

See Also

- SQL concepts: Data Type, Date and Time Constructs

- SQL time functions: CAST, CONVERT, CURTIME, HOUR, MINUTE, SECOND

- SQL timestamp functions: CURRENT_TIMESTAMP, GETDATE, GETUTCDATE, NOW, SYSDATE, TIMESTAM-PADD, TIMESTAMPDIFF

- InterSystems IRIS ObjectScript: $ZTIME function, $HOROLOG special variable, $ZTIMESTAMP special variable

CURRENT_TIMESTAMP (SQL)

A date/time function that returns the current local date and time.

Synopsis

```
CURRENT_TIMESTAMP CURRENT_TIMESTAMP(precision)
```

Arguments

Argument	Description
precision	A positive integer that specifies the time precision as the number of digits of fractional seconds. The default is 0 (no fractional seconds); this default is configurable.

CURRENT_TIMESTAMP returns the TIMESTAMP data type.

Description

CURRENT_TIMESTAMP takes either no arguments or a precision argument. Empty argument parentheses are not permitted.

CURRENT_TIMESTAMP returns the current local date and time for this timezone; it adjusts for local time variants, such as Daylight Saving Time.

CURRENT_TIMESTAMP can return a timestamp in either %TimeStamp data type format (yyyy-mm-dd hh:mm:ss.ffff) or %PosixTime data type format (an encoded 64-bit signed integer). The following rules determine which timestamp format is returned:

1. If the current timestamp is being supplied to a field of data type %PosixTime, the current timestamp value is returned in POSIXTIME data type format. For example, `WHERE PosixField=CURRENT_TIMESTAMP` or `INSERT INTO MyTable (PosixField) VALUES (CURRENT_TIMESTAMP)`.

2. If the current timestamp is being supplied to a field of data type %TimeStamp, the current timestamp value is returned in TIMESTAMP data type format. For example, `WHERE TSField=CURRENT_TIMESTAMP` or `INSERT INTO MyTable (TSField) VALUES (CURRENT_TIMESTAMP)`.

3. If the current timestamp is being supplied without context, the current timestamp value is returned in TIMESTAMP data type format. For example, `SELECT CURRENT_TIMESTAMP`.

You can use $HOROLOG to store or return the current local date and time in internal format.

To change the default datetime string format, use the SET OPTION command with the various date and time options.

You can specify **CURRENT_TIMESTAMP**, with or without *precision*, as the field default value when defining a datetime field using **CREATE TABLE** or **ALTER TABLE**. **CURRENT_TIMESTAMP** can be specified as the field default value for a field of data type %Library.PosixTime or %Library.TimeStamp; the current date and time is stored in the format specified by the field's data type.

Fractional Seconds Precision

CURRENT_TIMESTAMP has two syntax forms:

- Without argument parentheses, **CURRENT_TIMESTAMP** is functionally identical to NOW. It uses the system-wide default time precision.

- With argument parentheses, **CURRENT_TIMESTAMP(precision)**, is functionally identical to **GETDATE**, except that the **CURRENT_TIMESTAMP()** *precision* argument is mandatory. **CURRENT_TIMESTAMP()** always returns its specified *precision* and ignores the configured system-wide default time precision.

Fractional seconds are always truncated, not rounded, to the specified precision.

- In TIMESTAMP data type format, the maximum possible digits of precision is nine. The actual number of digits supported is determined by the *precision* argument, the configured default time precision, and the system capabilities. If you specify a *precision* larger than the configured default time precision, the additional digits of precision are returned as trailing zeros.

- In POSIXTIME data type format, the maximum possible digits of precision is six. Every POSIXTIME value is computed using six digits of precision; these fractional digits default to zeros unless supplied. The actual number of non-zero digits supported is determined by the *precision* argument, the configured default time precision, and the system capabilities.

Configuring Precision

The default precision can be configured using the following:

- SET OPTION with the TIME_PRECISION option.

- The system-wide **$SYSTEM.SQL.Util.SetOption()** method configuration option `DefaultTimePrecision`. To determine the current setting, call **$SYSTEM.SQL.CurrentSettings()** which displays `Default time precision`; the default is 0.

- Go to the Management Portal, select **System Administration**, **Configuration**, **SQL and Object Settings**, **SQL**. View and edit the current setting of **Default time precision for GETDATE(), CURRENT_TIME, and CURRENT_TIMESTAMP**.

Specify an integer 0 through 9 (inclusive) for the default number of decimal digits of precision to return. The default is 0. The actual precision returned is platform dependent; *precision* digits in excess of the precision available on your system are returned as zeroes.

Date and Time Functions Compared

GETDATE and NOW can also be used to return the current local date and time as either a TIMESTAMP data type or a POSIXTIME data type value. **GETDATE** supports precision, **NOW** does not support precision.

SYSDATE is identical to **CURRENT_TIMESTAMP**, with the exception that **SYSDATE** does not support *precision*. **CURRENT_TIMESTAMP** is the preferred InterSystems SQL function; **SYSDATE** is provided for compatibility with other vendors.

All InterSystems SQL time and date functions except GETUTCDATE are specific to the local time zone setting. To get a universal (time zone independent) timestamp, you can use either **GETUTCDATE** to return the universal date and time as either a TIMESTAMP data type or a POSIXTIME data type value, or the ObjectScript $ZTIMESTAMP special variable.

To return just the current local date, use CURDATE or CURRENT_DATE. To return just the current local time, use CURRENT_TIME or CURTIME. These functions return their values in DATE or TIME data type. None of these functions support precision.

The TIMESTAMP data type storage format and display format are the same. The POSIXTIME data type storage format is an encoded 64-bit signed integer. The TIME and DATE data types store their values as integers in $HOROLOG format; when displayed in SQL they are converted to date or time display format. Embedded SQL returns them in logical (storage) format by default. You can change the Embedded SQL returned value format using the #sqlcompile select macro preprocessor directive.

You can use the CAST or CONVERT function to change the data type of dates and times.

Examples

The following example returns the current local date and time three different ways: in TIMESTAMP data type format with system default time precision, with a *precision* of two digits of fractional seconds, and in $HOROLOG internal storage format with full seconds:

SQL

```
SELECT
   CURRENT_TIMESTAMP AS FullSecStamp,
   CURRENT_TIMESTAMP(2) AS FracSecStamp,
   $HOROLOG AS InternalFullSec
```

The following example sets a locale default time precision. The first **CURRENT_TIMESTAMP** specifies no *precision*; it returns the current time with the default time precision. The second **CURRENT_TIMESTAMP** specifies *precision*; this overrides the configured default time precision. The *precision* argument can be larger or smaller than the default time precision setting:

SQL

```
SELECT CURRENT_TIMESTAMP, CURRENT_TIMESTAMP(2)
```

The following example compares local (time zone specific) and universal (time zone independent) time stamps:

SQL

```
  SELECT CURRENT_TIMESTAMP,GETUTCDATE()
```

The following example sets the LastUpdate field in the selected row of the Orders table to the current system date and time. If LastUpdate is data type %TimeStamp, **CURRENT_TIMESTAMP** returns the current date and time as an ODBC timestamp; if LastUpdate is data type %PosixTime, **CURRENT_TIMESTAMP** returns the current date and time as an encoded 64-bit signed integer:

SQL

```
UPDATE Orders SET LastUpdate = CURRENT_TIMESTAMP
  WHERE Orders.OrderNumber=:ord
```

The following example creates a table named Orders, which records product orders received:

SQL

```
CREATE TABLE Orders (
     OrderId     INT NOT NULL,
     ClientId    INT,
     ItemName    CHAR(40) NOT NULL,
     OrderDate   TIMESTAMP DEFAULT CURRENT_TIMESTAMP(3),
     PRIMARY KEY (OrderId))
```

The OrderDate column contains the date and time that the order was received. It uses the TIMESTAMP data type and inserts the current system date and time as the default value using the **CURRENT_TIMESTAMP** function with a precision of 3.

See Also

- SQL concepts: Data Type, Date and Time Constructs

- SQL timestamp functions: CAST, CONVERT, GETDATE, GETUTCDATE, NOW, SYSDATE, TIMESTAMPADD, TIMESTAMPDIFF, TO_POSIXTIME, TO_TIMESTAMP

- SQL current date and time functions: CURDATE, CURRENT_DATE, CURRENT_TIME, CURTIME

- ObjectScript: $ZDATETIME function, $HOROLOG special variable, $ZTIMESTAMP special variable

CURTIME (SQL)

A scalar date/time function that returns the current local time.

Synopsis

```
{fn CURTIME()}
{fn CURTIME}
```

Description

CURTIME returns the current local time as data type TIME. It takes no argument; note that the argument parentheses are optional. **CURTIME** returns the current local time for this timezone; it adjusts for local time variants, such as Daylight Saving Time.

CURTIME in Logical mode returns the current local time in $HOROLOG format; for example, 37065. **CURTIME** in Display mode returns the current local time in the default format for the locale; for example, 10:18:27.

Hours are represented in 24-hour format.

To change the default time format, use the SET OPTION command with the TIME_FORMAT and TIME_PRECISION options.

To return just the current time, use **CURTIME** or CURRENT_TIME. These functions return their values in TIME data type. The CURRENT_TIMESTAMP, GETDATE and NOW functions can also be used to return the current date and time as a TIMESTAMP data type.

Note that all InterSystems SQL time and date functions except GETUTCDATE are specific to the local time zone setting. To get a current timestamp that is universal (independent of time zone) you can use **GETUTCDATE** or the ObjectScript $ZTIMESTAMP special variable.

These data types perform differently when using embedded SQL. The TIME data type stores values as integers in $HOROLOG format (as the number of seconds since midnight); when displayed in SQL they are converted to time display format; when returned from embedded SQL they are returned as integers. A TIMESTAMP data type stores and displays its value in the same format. You can use the CAST or CONVERT function to change the data type of times and dates.

Examples

The following examples both return the current system time:

SQL

```
SELECT {fn CURTIME()} AS TimeNow
```

SQL

```
SELECT {fn CURTIME} AS TimeNow
```

The following Embedded SQL example returns the current time. Because this time is stored in $HOROLOG format, it is returned as an integer:

ObjectScript

```
  &sql(SELECT {fn CURTIME} INTO :a)
  IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
  ELSE {
    WRITE !,"Current time is: ",a }
```

The following example sets the LastCall field in the selected row of the Contacts table to the current system time:

SQL

```
UPDATE Contacts Set LastCall = {fn CURTIME()}
  WHERE Contacts.ItemNumber=:item
```

See Also

- SQL concepts: Data Type Date and Time Constructs

- SQL time functions: CAST CONVERT CURRENT_TIME HOUR MINUTE SECOND

- SQL timestamp functions: CURRENT_TIMESTAMP GETDATE GETUTCDATE NOW TIMESTAMPADD TIMESTAMPDIFF

- ObjectScript: $ZTIME function $HOROLOG special variable $ZTIMESTAMP special variable

DATABASE

A scalar string function that returns the database name qualifier.

Synopsis

```
{fn DATABASE()}
```

Description

DATABASE returns the current qualifier for the name of the database corresponding to the connection handle. In InterSystems IRIS, **DATABASE** always returns the empty string (").

DATALENGTH (SQL)

A function that returns the number of characters in an expression.

Synopsis

```
DATALENGTH(expression)
```

Description

The **DATALENGTH** returns the number of characters used for an expression.

The **DATALENGTH**, **CHAR_LENGTH**, and **CHARACTER_LENGTH** functions are identical. Use of the **CHAR_LENGTH** function is recommended for new code. **DATALENGTH** is provided for TSQL compatibility. Refer to CHAR_LENGTH for further details.

Arguments

expression

An expression, which can be the name of a column, a string literal, or the result of another scalar function. The underlying data type can be a character type (such as CHAR or VARCHAR), a numeric, or a data stream.

DATALENGTH returns the INTEGER data type.

See Also

*   CHAR_LENGTH function

*   CHARACTER_LENGTH function

DATE (SQL)

A function that takes a timestamp and returns a date.

Synopsis

```
DATE(timestamp)
```

Description

- **DATE(*timestamp*)** takes a timestamp expression and returns a date of data type DATE. This function is equivalent to CAST(*timestamp* as DATE).

  This statement converts a timestamp string to a date. The time portion of the timestamp is validated but not returned.

  SQL

  ```
  SELECT DATE('2000-01-01 00:00:00') AS StringToDate -- Display Mode: 01/01/2000
  ```

  Example: Convert Timestamps of Varying Types to Dates

Arguments

timestamp

An expression that specifies a timestamp, date, or date and time representation. Specify *timestamp* as one of these data type classes:

- %Library.TimeStamp

- %Library.PosixTime

- %Library.Date

- %Library.Integer

- %Library.Numeric values in implicit logical timestamp format, such as +**$HOROLOG**. See Numeric Timestamps.

- %Library.String values compatible with %Library.TimeStamp. See String Timestamps.

Numeric Timestamps

The **$HOROLOG** and **$ZTIMESTAMP** special variables specify the current date as a numeric character string. **DATE** casts such strings to 0, which represents the date December 31, 1840.

SQL

```
SELECT
  DATE($HOROLOG),   -- Returns 0 (12/31/1840)
  DATE($ZTIMESTAMP) -- Returns 0 (12/31/1840)
```

To interpret **$HOROLOG** or **$ZTIMESTAMP** as the current date, you must force numeric interpretation by prefixing the date with a plus (+) sign.

SQL

```
SELECT
  DATE(+$HOROLOG),   -- Returns the current date
  DATE(+$ZTIMESTAMP) -- Returns the current date
```

String Timestamps

String timestamps converted to the DATE format, must be compatible with the %Library.TimeStamp data type. The data type stores strings in the ODBC date format, and the **DATE** function validates input strings against this format. If the string passes validation, **DATE** returns the corresponding date. If it fails validation, **DATE** returns 0, which corresponds to the date December 31, 1840. An empty string (") also returns 0. A NULL argument returns NULL.

DATE performs these validation checks:

- The string corresponds to ODBC format, where `fff` is fractional seconds:

  ```
  yyyy-mm-dd hh:mm:ss.fff
  ```

- The string contains at least a full date: `yyyy-mm-dd`. The time portion is optional. **DATE** validates any specified times but does not return them. Any part of the time can be included. For example: `yyyy-mm-dd hh:`

- The string does not contain any invalid format characters or trailing characters. Leading zeros can be omitted or included.

- Each numeric element of the string, including the optional time portion, contains a valid value given the range for that element. For example:

  – Month values are from 1 to 12.

  – Day values do not exceed the number of days for the specified month (includes leap year days).

  – Dates are within the range of `0001-01-01` to `9999-12-31`.

Examples

Convert Timestamps of Varying Formats to Dates

These statements convert timestamps of data type %Library.TimeStamp to the DATE data type.

SQL

```
SELECT
  {fn NOW} AS NowCol,
  DATE({fn NOW}) AS DateCol
```

SQL

```
SELECT
  CURRENT_TIMESTAMP AS TSCol,
  DATE(CURRENT_TIMESTAMP) AS DateCol
```

SQL

```
SELECT
  GETDATE() AS GetDateCol,
  DATE(GETDATE()) AS DateCol
```

These statements convert strings written in the %Library.TimeStamp format to DATE.

SQL

```
SELECT
  '2022-05-22 13:14:23' AS DateStrCol,
  DATE('2022-05-22 13:14:23') AS DateCol
```

This statement converts a %Library.PosixTime timestamp to DATE.

SQL

```
SELECT
  TO_POSIXTIME('2022-05-22','YYYY-MM-DD') AS PosixCol,
  DATE(TO_POSIXTIME('2022-05-22','YYYY-MM-DD')) AS DateCol
```

These statements convert string values that represent dates in the InterSystems IRIS® logical format to DATE. To convert these values properly, you must force numeric evaluation by prefixing the strings with a plus sign (+).

SQL

```
SELECT
  $H AS HoroCol,
  DATE(+$H) AS DateCol
```

SQL

```
SELECT
  $ZTIMESTAMP AS TSCol,
  DATE(+$ZTIMESTAMP) AS DateCol
```

Alternatives

To perform equivalent timestamp-to-date conversions in ObjectScript using code, use the **DATE()** method:

ObjectScript

```
WRITE $SYSTEM.SQL.Functions.DATE("2018-02-23 12:37:45")
```

See Also

• CAST function

• CURDATE and CURRENT_DATE functions

• CURRENT_TIMESTAMP function

• GETUTCDATE function

• NOW function

• TO_TIMESTAMP function

• $HOROLOG special variable

• $ZTIMESTAMP special variable

DATEADD (SQL)

A date/time function that returns a timestamp calculated by adding or subtracting a number of date part units (such as hours or days) to a date or timestamp.

Synopsis

```
DATEADD(datePart,numUnits,date)
```

Description

- **DATEADD(*datePart*,*numUnits*,*date*)** modifies a date or time expression by incrementing the specified date part by the specified number of units. If you specify negative units, then **DATEADD** decrements the date by that number of units.

    - If *date* is of type %Library.PosixTime (an encoded 64-bit signed integer), then **DATEADD** returns a timestamp of type %Library.PosixTime.

    - If *date* is of any other type, then **DATEADD** returns a timestamp of type %Library.TimeStamp in the format yyyy-mm-dd hh:mm:ss.fff.

    This statement increments the current date by 5 months.

    ### SQL

    ```
    SELECT DATEADD('month',5,CURRENT_DATE)
    ```

DATEADD is compatible with Sybase and Microsoft SQL Server.

Arguments

datePart

The full or abbreviated name of a date or time part. You can specify *datePart* in uppercase or lowercase. In embedded SQL, specify *datePart* as a literal value or host variable. This table shows the valid date and time part names and abbreviations. It also shows by how much a single unit of that part (*numUnits* = 1) increments the date.

Name	Abbreviations	*numUnits* = 1
year	YYYY, YY	Increments year by 1.
quarter	qq, q	Increments month by 3.
month	mm, m	Increments month by 1.
week	wk, ww	Increments day by 7.
weekday	dw	Increments day by 1.
day	dd, d	Increments day by 1.
dayofyear	dy	Increments day by 1.
hour	hh	Increments hour by 1.
minute	mi, n	Increments minute by 1.
second	ss, s	Increments second by 1.

Name	Abbreviations	*numUnits* = 1
millisecond	ms	Increments second by 0.001 (precision of 3)
microsecond	mcs	Increments second by 0.000001 (precision of 6)
nanosecond	ns	Increments second by 0.000000001 (precision of 9)

Incrementing or decrementing a date part causes other date parts to be modified appropriately. For example, incrementing the hour past midnight automatically increments the day, which may in turn increment the month, and so forth.

DATEADD performs slightly different operations depending on whether you specify *datePart* with or without quotes:

- Quotes: DATEADD('month',1,'2022-02-25'): *datePart* is treated as a literal. When processing the query, InterSystems SQL performs literal substitution, replacing the 'month' string with an input parameter, which produces a more generally reusable cached query.

- No quotes: DATEADD(month,1,'2022-02-25'): *datePart* is treated as a keyword. When processing the query, InterSystems SQL does not perform literal substitution, which produces a more specific cached query.

Specifying an invalid *datePart* literal value generates an SQLCODE -8 error code. However, if you supply an invalid *datePart* value as a host variable, no SQLCODE error is issued and the **DATEPART** function that is called to parse *datePart* returns a value of NULL.

numUnits

The number of *datePart* units added to or subtracted from *date*, specified as a numeric value. **DATEADD** truncates *numUnits* to an integer. If *numUnits* contains no numeric parts or does not start with a numeric value, **DATEADD** truncates this value to 0 and returns the originally specified *date*.

date

The date or time expression being added to or subtracted from, specified as one of these InterSystems IRIS® data types:

- %Date logical value (+$H), also known as $HOROLOG format.

- %PosixTime (%Library.PosixTime) logical value (an encoded 64-bit signed integer).

- %TimeStamp (%Library.TimeStamp) logical value (YYYY-MM-DD HH:MM:SS.FFF), also known as ODBC format.

- %String or string-compatible value, which can be in one of these formats:

  Table G–1: $HOROLOG Date and Time Format

Format	Example
ddddd	SELECT DATEADD('yy',1,'66716')
ddddd,sssss	SELECT DATEADD('yy',1,'66716,256')
ddddd,sssss.fff	SELECT DATEADD('yy',1,'66716,256.467')

  where:

  – ddddd is the integer number of days since December 31, 1840.

  – sssss is the integer number of seconds since the start of that day.

  – fff is the integer number of fractional seconds. If you specify fractional seconds, the returned **DATEADD** value also includes fractional seconds.

Table G–2: Date Format

Format	Example
MM/DD/YY	`SELECT DATEADD('year',1,'12/31/99')`
MM/DD/YYYY	`SELECT DATEADD('year',1,'8/24/2022')`
MM-DD-YY	`SELECT DATEADD('year',1,'12-31-99')`
MM-DD-YYYY	`SELECT DATEADD('year',1,'8-24-2022')`
MM.DD.YY	`SELECT DATEADD('year',1,'12.31.99')`
MM.DD.YYYY	`SELECT DATEADD('year',1,'8.24.2022')`
Mmm DD YY	`SELECT DATEADD('year',1,'Dec 30 92')`
Mmm DD YYYY	`SELECT DATEADD('year',1,'January 23 2021')`
Mmm DD, YY	`SELECT DATEADD('year',1,'Dec 30, 92')`
Mmm DD, YYYY	`SELECT DATEADD('year',1,'January 23, 2021')`

where:

– MM is the two-digit month.

– DD is the two-digit number of days in the month.

– Mmm is the spelled-out month. You can specify a minimum of three letters (for example, `Mar`) up to the full month name (for example, `March`).

– YY and YYYY are the two-digit and four-digit forms of the year, respectively.

You can specify *date* as a combined date and time string. For example:

SQL

```
SELECT DATEADD('hh',1,'12/22/2021 8:15:23')
```

If you specify a time without a date, **DATEADD** defaults to date 01/01/1900.

Table G–3: Time Format

Format	Example
HH:	`SELECT DATEADD('hour',1,'10:')`
HH:MM	`SELECT DATEADD('mi',1,'10:30')`
HH:MM:SS	`SELECT DATEADD('ss',1,'10:30:59')`
HH...SS.FFF	`SELECT DATEADD('ms',1,'10:30:59.245')`
HH...[AM|PM]	`SELECT DATEADD('mi',1,'10:30PM')`

where:

– HH is the two-digit number of hours into the day.

– MM is the two-digit number of minutes into the hour.

- – SS is the two-digit number of seconds into the minute.

- – FFF is the number of fractional seconds.

You can specify *date* as a combined date and time string. For example:

SQL

```
SELECT DATEADD('hh',1,'12/22/2021 8:15:23')
```

If you specify a date without a time, **DATEADD** defaults to time 00:00:00.

The *date* argument has these restrictions and behaviors:

- The date string must be complete and properly formatted with the appropriate number of elements and digits for each element and the appropriate separator character. Years must be specified as four digits. If you omit the date portion of an input value, **DATEADD** defaults to '1900-01-01'.

- Date and time values must be within the valid range:

  - – Years — 0001 through 9999

  - – Months — 1 through 12

  - – Days — 1 through 31

  - – Hours — 00 through 23

  - – Minutes — 0 through 59

  - – Seconds — 0 through 59

  The number of days in a month must match the month and year. For example, the date '02-29' is valid only if the specified year is a leap year.

- In date values less than 10 (month and day) a leading zero in optional. Other non-canonical integer values are not permitted. For example, a `Day` value of `'07'` or `'7'` is valid, but `'007'`, `'7.0'`, or `'7a'` are not valid.

- Time values are optional. If *date* specifies an incomplete time, zeros are supplied for the unspecified parts.

- An hour value less than 10 must include a leading zero.

Examples

Add Varying Time Units to Dates

This statement adds 1 week to the specified date. It returns 2022-03-05 00:00:00, because adding 1 week adds 7 days. **DATEADD** supplies the omitted time portion.

SQL

```
SELECT DATEADD('week',1,'2022-02-26') AS NewDate
```

The statement adds 5 months to the specified timestamp and returns 2022-04-26 12:00:00. **DATEADD** modifies both the month and year, because adding 5 months also increments the year.

SQL

```
SELECT DATEADD(MM,5,'2021-11-26 12:00:00') AS NewDate
```

This statement also adds 5 months to the timestamp and returns 2021-06-30 12:00:00. **DATEADD** modifies both the day and month, because incrementing only the month results in an invalid date of June 31.

SQL

```
SELECT DATEADD('mm',5,'2021-01-31 12:00:00') AS NewDate
```

This statement adds 45 minutes to the timestamp and returns 2022-02-26 12:45:00.

SQL

```
SELECT DATEADD(MI,45,'2022-02-26 12:00:00') AS NewTime
```

This statement also adds 45 minutes to the timestamp, but in this case the addition increments the day, which increments the month. It returns 2022-03-01 00:15:00.

SQL

```
SELECT DATEADD('mi',45,'2022-02-28 23:30:00') AS NewTime
```

This statement decrements the original timestamp by 45 minutes and returns 2021-12-31 23:25:00.

SQL

```
SELECT DATEADD(N,-45,'2022-01-01 00:10:00') AS NewTime
```

This statement adds 60 days to the current date, adjusting for the varying lengths of months.

SQL

```
SELECT DATEADD(D,60,CURRENT_DATE) AS NewDate
```

The first **DATEADD** of this statement adds 92 days to the specified date and returns 2022-03-22 00:00:00. The second **DATEADD** adds 1 quarter to the specified date and returns 2022-03-20 00:00:00. Incrementing by a quarter increments the month field by 3. If necessary, **DATEADD** also increments the year field and corrects for the maximum number of days for a given month.

SQL

```
SELECT DATEADD('dd',92,'2021-12-20') AS NewDateD,
       DATEADD('qq',1,'2021-12-20') AS NewDateQ
```

The previous statements all use date part abbreviations. However, you can also specify the date part by its full name. For example, this statement adds 92 days to the date and returns 2022-03-22 00:00:00.

SQL

```
SELECT DATEADD('day',92,'2021-12-20') AS NewDate
```

This Embedded SQL code uses host variables to perform the same **DATEADD** operation as the previous SQL statement.

ObjectScript

```
set datePart = "day"
set numUnits = 92
set dateIn = "2021-12-20"

&sql(SELECT DATEADD(:datePart,:numUnits,:dateIn) INTO :dateOut)

write "in: ",dateIn,!,"out: ",dateOut
```

Alternatives

You can perform time and date modifications using the TIMESTAMPADD ODBC scalar function.

To perform equivalent timestamp conversions in ObjectScript, use the **DATEADD()** method:

`$SYSTEM.SQL.Functions.DATEADD(datePart,numUnits,date)`

See Also

- DATEDIFF function
- DATENAME function
- DATEPART function
- TIMESTAMPADD function
- TIMESTAMPDIFF function

DATEDIFF (SQL)

A date/time function that returns an integer difference for a specified datepart between two dates.

Synopsis

```
DATEDIFF(datePart,startDate,endDate)
```

Description

- **DATEDIFF**(*datePart*,*startDate*,*endDate*) returns an INTEGER value that is the difference between the starting and ending date (*startDate* minus *endDate*) for the specified date part (seconds, days, weeks, and so on). If *endDate* is earlier than *startDate*, **DATEDIFF** returns a negative INTEGER value.

  This statement returns 353 because there are 353 days (D) between the two timestamps:

  SQL

  ```
  SELECT DATEDIFF(D,'2022-01-01 00:00:00','2022-12-20 12:00:00')
  ```

  Example: Calculate Differences Between Dates

DATEDIFF is compatible with Sybase and Microsoft SQL Server. You can perform similar time and date comparisons using the TIMESTAMPDIFF ODBC scalar function.

Arguments

datePart

The name or abbreviated name of a date or time part. You can specify *datePart* in uppercase or lowercase. In embedded SQL, you specify *datePart* as a literal value or host variable. This table shows the valid date and time parts:

Name	Abbreviations
year	yyyy, yy
month	mm, m
week	wk, ww
weekday	dw
day	dd, d
dayofyear	dy
hour	hh
minute	mi, n
second	ss, s
millisecond	ms
microsecond	mcs
nanosecond	ns

The weekday and dayofyear date part values are functionally identical to the day date part value.

DATEDIFF does not handle quarters (3-month intervals).

If you specify start and end dates that include fractional seconds, **DATEDIFF** returns the difference as an integer number of fractional seconds. For example:

SQL

```
SELECT DATEDIFF('ms','64701,56670.10','64701,56670.27'),     /* returns 170 */
       DATEDIFF('ms','64701,56670.1111','64701,56670.27222') /* returns 161 */
```

DATEDIFF returns fractional seconds as milliseconds (a three-digit integer), microseconds (6-digit integer), or nanoseconds (9-digit integer) regardless of the number of fractional digits precision in *startDate* and *endDate*. For example:

DATEDIFF performs slightly different operations depending on whether you specify *datePart* with or without quotes:

- Quotes: DATEDIFF('month','2018-02-25',$HOROLOG): *datePart* is treated as a literal. When processing the query, InterSystems SQL performs literal substitution, replacing the 'month' string with an input parameter, which produces a more generally reusable cached query.

- No quotes: DATEDIFF(month,'2018-02-25',$HOROLOG): *datePart* is treated as a keyword. When processing the query, InterSystems SQL does not perform literal substitution, which produces a more specific cached query.

In Embedded SQL, specifying an invalid *datePart* as an input variable returns an SQLCODE -8 error. Specifying an invalid *datePart* as a literal value returns a <SYNTAX> error.

In Dynamic SQL, if you supply an invalid *datepart*, **DATEDIFF** returns NULL. No SQLCODE error is issued.

startDate,endDate

The starting and ending dates over which **DATEDIFF** calculates the difference, specified as one of these InterSystems IRIS® data types:

- %Date logical value (+$H), also known as $HOROLOG format.

- %PosixTime (%Library.PosixTime) logical value (an encoded 64-bit signed integer).

- %TimeStamp (%Library.TimeStamp) logical value (YYYY-MM-DD HH:MM:SS.FFF), also known as ODBC format.

- %String or string-compatible value, which can be in one of these formats:

Table G–4: $HOROLOG Date and Time Format

Format	Example
ddddd	SELECT DATEDIFF('yy','65726','66716')
ddddd,sssss	SELECT DATEDIFF('mi','65726,143','66716,256')
ddddd,sssss.fff	SELECT DATEDIFF('ns','65726,143.345','66716,256.467')

where:

- ddddd is the integer number of days since December 31, 1840.

- sssss is the integer number of seconds since the start of that day.

- fff is the integer number of fractional seconds.

Table G–5: Date Format

Format	Example
MM/DD/YY	SELECT DATEDIFF('yy','11/25/80','12/31/99')
MM/DD/YYYY	SELECT DATEDIFF('dd','3/15/2017','8/24/2022')
MM-DD-YY	SELECT DATEDIFF('yy','11-25-80','12-31-99')
MM-DD-YYYY	SELECT DATEDIFF('dd','3-15-2017','8-24-2022')
MM.DD.YY	SELECT DATEDIFF('yy','11.25.80','12.31.99')
MM.DD.YYYY	SELECT DATEDIFF('dd','3.15.2017','8.24.2022')
Mmm DD YY	SELECT DATEDIFF('ss','Sep 9 91','Dec 30 92')
Mmm DD YYYY	SELECT DATEDIFF('ss','October 10 2019','January 23 2021')
Mmm DD, YY	SELECT DATEDIFF('ss','Sep 9, 91','Dec 30, 92')
Mmm DD, YYYY	SELECT DATEDIFF('ss','October 10, 2019','January 23, 2021')

where:

– MM is the two-digit month.

– DD is the two-digit number of days in the month.

– Mmm is the spelled-out month. You can specify a minimum of three letters (for example, Mar) up to the full month name (for example, March).

– YY and YYYY are the two-digit and four-digit forms of the year, respectively.

You can specify *startDate* and *endDate* as combined date and time strings. For example:

SQL

```
SELECT DATEDIFF('hh','12/22/2021 8:15:23','12/31/2021 10:30:23')
```

If you specify a time without a date, **DATEDIFF** defaults to date 01/01/1900.

Table G–6: Time Format

Format	Example
HH:	SELECT DATEDIFF('hh','2:','10:')
HH:MM	SELECT DATEDIFF('mi','2:15','10:30')
HH:MM:SS	SELECT DATEDIFF('ss','2:15:23','10:30:59')
HH...SS:FFF	SELECT DATEDIFF('ms','2:15:23:335','10:30:59:245')
HH...SS.FFF	SELECT DATEDIFF('ms','2:15:23.335','10:30:59.245')
HH...[AM|PM]	SELECT DATEDIFF('mi','2:15AM','10:30PM')

where:

– HH is the two-digit number of hours into the day.

– MM is the two-digit number of minutes into the hour.

– SS is the two-digit number of seconds into the minute.

– FFF is the number of fractional seconds.

You can specify *startDate* and *endDate* as combined date and time strings. For example:

SQL

```
SELECT DATEDIFF('hh','12/22/2021 8:15:23','12/31/2021 10:30:23')
```

If you specify a date without a time, **DATEDIFF** defaults to time 00:00:00.

You can specify *startDate* and *endDate* arguments in different data types.

startDate and *endDate* arguments have these restrictions and behaviors:

• A date string must be complete and properly formatted with the appropriate number of elements and digits for each element and the appropriate separator character. Years must be specified as four digits. If you omit the date portion of an input value, **DATEDIFF** defaults to '1900-01-01'.

• Date and time values must be within the valid range:

– Years — 0001 through 9999

– Months — 1 through 12

– Days — 1 through 31

– Hours — 00 through 23

– Minutes — 0 through 59

– Seconds — 0 through 59

The number of days in a month must match the month and year. For example, the date '02-29' is valid only if the specified year is a leap year. An invalid date value results in an SQLCODE -8 error.

• In date values less than 10 (month and day) a leading zero in optional. Other non-canonical integer values are not permitted. For example, a Day value of '07' or '7' is valid, but '007', '7.0', or '7a' are not valid.

- Time values are optional. If *startDate* or *endDate* specifies an incomplete time, zeros are supplied for the unspecified parts.

- An hour value less than 10 must include a leading zero. Omitting this leading zero results in an SQLCODE -8 error.

- In Embedded SQL, specifying an invalid *startDate* or *endDate* as either an input variable or literal returns an SQLCODE -8 error.

- In Dynamic SQL, if you supply an invalid *startDate* or *endDate*, **DATEDIFF** returns NULL. No SQLCODE error is issued.

- Two-digit years from 01 to 99 are assumed to be from 1901 to 1999. For example, in this statement, the *startDate* year is 1914:

SQL

```
SELECT DATEDIFF('year','10/11/14','04/22/2022'),
       DATEDIFF('year','12:00:00','2022-04-22 12:00:00')
```

Specifying 00 is treated as year 0000, which is invalid and returns an error.

To change the default sliding window that controls this date system-wide, use the **^%DATE** legacy utility. For information on establishing a sliding window for interpreting a specified date with a two-digit year, see the $ZDATE, $ZDATEH, $ZDATETIME and $ZDATETIMEH functions.

Examples

Calculate Differences Between Dates

DATEDIFF returns the total number of the specified unit between *startDate* and *endDate*. For example, this statement calculates the number of minutes between dates. It evaluates both the date and time components. For each day difference, it adds 1440 minutes, which is the number of minutes in a day.

SQL

```
SELECT DATEDIFF('mi','1910-08-21 08:32:04','1910-08-28 01:45:00')
```

DATEDIFF does not account for the actual duration between dates. In this way, it can be considered as a count of the specified date part boundaries crossed between *startDate* and *endDate*. For example, these differences between consecutive years all return a **DATEDIFF** of 1, even though their durations are greater than or less than 365 days.

SQL

```
SELECT DATEDIFF('yyyy','1910-08-21','1911-08-21') AS ExactYear,
       DATEDIFF('yyyy','1910-06-30','1911-01-01') AS HalfYear,
       DATEDIFF('yyyy','1910-01-01','1911-12-31') AS Nearly2Years,
       DATEDIFF('yyyy','1910-12-31 11:59:59','1911-01-01 00:00:00') AS NewYearSecond
```

Similarly, in this statement, the difference these consecutive minutes is 1, even though only 6 seconds actually separate the two values.

SQL

```
SELECT DATEDIFF('mi','12:23:59','12:24:05') AS MinuteDiff
```

The previous statements used abbreviations for the date part. Alternatively, you can specify the full name of the date part. For example:

SQL

```
SELECT DATEDIFF('minute','12:23:59','12:24:05') AS MinuteDiff
```

This Embedded SQL example uses host variables to perform the same **DATEDIFF** operation as the previous statement:

ObjectScript

```
set datePart="minute"
set startDate="12:23:59"
set endDate="12:24:05"

&sql(SELECT DATEDIFF(:datePart,:startDate,:endDate) INTO :diff)
WRITE diff, !
```

This statement uses **DATEDIFF** in the WHERE clause to select patients admitted in the last week:

SQL

```
SELECT Name,DateOfAdmission FROM Sample.Patients WHERE DATEDIFF(D,DateOfAdmission,$HOROLOG) <= 7
```

This statement uses a subquery to return those records where the person's date of birth is 1500 days or fewer from the current date:

SQL

```
SELECT Name,Age,DOB
FROM (SELECT Name,Age,DOB, DATEDIFF('dy',DOB,$HOROLOG) AS DaysTo FROM Sample.Person)
WHERE DaysTo <= 1500
ORDER BY Age
```

Time Differences Independent of Time Format

DATEDIFF returns a time difference in seconds and milliseconds, even when the time format for the current process is set to not return seconds. For example:

ObjectScript

```
// Get current time format and set start and end times
set originalTimeFormat = ##class(%SYS.NLS.Format).GetFormatItem("TimeFormat")
set startDate = "66211,34717.10"
set endDate = "66211,34720.27"

// Set time format that includes seconds (TimeFormat = 1)
do ##class(%SYS.NLS.Format).SetFormatItem("TimeFormat",1)
write "DATETIME (with seconds): ",!, $ZDATETIME(startDate,1,-1)," ",$ZDATETIME(endDate,1,-1),!
&sql(SELECT DATEDIFF('ss',:startDate,:endDate) INTO :diff)
write "DATEDIFF number of seconds: ",diff,!!

// Set time format that omits seconds (TimeFormat = 2)
do ##class(%SYS.NLS.Format).SetFormatItem("TimeFormat",2)
write "DATETIME (without seconds): ",!, $ZDATETIME(startDate,1,-1)," ",$ZDATETIME(endDate,1,-1),!
&sql(SELECT DATEDIFF('ss',:startDate,:endDate) INTO :diff)
write "DATEDIFF number of seconds: ",diff,!

// Revert to original time format
do ##class(%SYS.NLS.Format).SetFormatItem("TimeFormat",originalTimeFormat)
```

Alternatives

To call this function in ObjectScript code, use the **DATEDIFF()** method:

```
$SYSTEM.SQL.Functions.DATEDIFF(datePart,startDate,endDate)
```

Specifying an invalid *datepart* to the **DATEDIFF()** method generates a <ZDDIF> error.

See Also

- DATEADD

- DATENAME

- DATEPART

- [TIMESTAMPADD](#)

- [TIMESTAMPDIFF](#)

- **^%DATE** legacy documentation at https://docs.intersystems.com/priordocexcerpts

DATENAME (SQL)

A date/time function that returns a string representing the value of the specified part of a date/time expression.

Synopsis

```
DATENAME(datepart,date-expression)
```

Description

The **DATENAME** function returns the name of the specified part (such as the month "June") of a date/time value. The result is returned as data type VARCHAR(20). If the result is numeric (such as "23" for the day), it is still returned as a VARCHAR(20) string. To return this information as an integer, use DATEPART. To return a string containing multiple date parts, use TO_DATE.

Note that **DATENAME** is provided for Sybase and Microsoft SQL Server compatibility.

This function can also be invoked from ObjectScript using the **DATENAME()** method call:

```
$SYSTEM.SQL.Functions.DATENAME(datepart,date-expression)
```

Datepart Argument

The *datepart* argument can be a string containing one (and only one) of the following date/time components, either the full name (the Date Part column) or its abbreviation (the Abbreviation column). These *datepart* component names and abbreviations are not case-sensitive.

Date Part	Abbreviations	Return Values
year	yyyy, yy	0001-9999
quarter	qq, q	1-4
month	mm	January,...December
week	wk, ww	1-53
weekday	dw	Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
dayofyear	dy, y	1-366
day	dd, d	1-31
hour	hh	0-23
minute	mi, n	0-59
second	ss, s	0-59
millisecond	ms	0-999 (with precision of 3)
microsecond	mcs	0–999999 (with precision of 6)
nanosecond	ns	0–999999999 (with precision of 9)

If you specify an invalid *datepart* value as a literal, an SQLCODE -8 error code is issued. However, if you supply an invalid *datepart* value as a host variable, no SQLCODE error is issued and the **DATENAME** function returns a value of NULL.

The preceding table shows the default return values for the various date parts. You can modify the returned values for several of these date parts by using the SET OPTION command with various time and date options.

`week`: InterSystems IRIS can be configured to determine the week of the year for a given date using either the InterSystems IRIS default algorithm or the ISO 8601 standard algorithm. For further details, refer to the WEEK function.

`weekday`: The InterSystems IRIS default for weekday is to designate Sunday as first day of the week (weekday=1). However, you can configure the first day of the week to another value, or you can apply the ISO 8601 standard which designates Monday as first day of the week. For further details, refer to the DAYOFWEEK function.

`millisecond`: InterSystems IRIS returns a string containing the number of milliseconds (thousandths of a second). If the *date-expression* has more than three fractional digits of precision, InterSystems IRIS truncates it to three digits and returns this number as a string. If the *date-expression* has a specified precision, but less than three fractional digits of precision, InterSystems IRIS zero pads it to three digits and returns this number as a string. `microsecond` and `nanosecond` perform similar truncation and zero padding.

A *datepart* can be specified as a quoted string or without quotes. These syntax variants perform slightly different operations:

- Quotes: `DATENAME('month','2018-02-25')`: the *datepart* is treated as a literal when creating cached queries. InterSystems SQL performs literal substitution. This produces a more generally reusable cached query.

- No quotes: `DATENAME(month,'2018-02-25')`: the *datepart* is treated as a keyword when creating cached queries. No literal substitution. This produces a more specific cached query.

Date Expression Formats

The *date-expression* argument can be in any of the following formats:

- An InterSystems IRIS %Date logical value (+$H)

- An InterSystems IRIS %PosixTime (%Library.PosixTime) logical value (an encoded 64-bit signed integer)

- An InterSystems IRIS %TimeStamp (%Library.TimeStamp) logical value (YYYY-MM-DD HH:MM:SS.FFF), also known as ODBC format.

- An InterSystems IRIS %String (or compatible) value

The InterSystems IRIS %String (or compatible) value can be in any of the following formats:

- 99999,99999 ($H format)

- *Sybase/SQL-Server-date Sybase/SQL-Server-time*

- *Sybase/SQL-Server-time Sybase/SQL-Server-date*

- *Sybase/SQL-Server-date* (default time is 00:00:00)

- *Sybase/SQL-Server-time* (default date is 01/01/1900)

Sybase/SQL-Server-date is one of these five formats:

`mm`*delimiter*`dd`*delimiter*`[yy]yy dd Mmm[mm][,][yy]yy dd [yy]yy Mmm[mm] yyyy Mmm[mm] dd yyyy [dd] Mmm[mm]`

where *delimiter* is a slash (/), hyphen (-), or period (.).

If the year is given as two digits, InterSystems IRIS checks the sliding window to interpret the date. The system default for the sliding window can be set via the legacy **^%DATE** legacy utility. For information on setting the sliding window for the current process, see the ObjectScript $ZDATE, $ZDATEH, $ZDATETIME and $ZDATETIMEH functions.

Sybase/SQL-Server-time represents one of these three formats:

`HH:MM[:SS:SSS][{AM|PM}] HH:MM[:SS.S] HH[''] {AM|PM}`

If the *date-expression* specifies a time format but does not specify a date format, **DATENAME** defaults to the date 1900–01–01, which has a weekday value of Monday.

Range and Value Checking

DATENAME performs the following checks on input values. If a value fails a check, the null string is returned.

- A valid *date-expression* may consist of a date string (yyyy-mm-dd), a time string (hh:mm:ss), or a date and time string (yyyy-mm-dd hh:mm:ss). If both date and time are specified, both must be valid. For example, you can return a Year value if no time string is specified, but you cannot return a Year value if an invalid time string is specified.

- A date string must be complete and properly formatted with the appropriate number of elements and digits for each element, and the appropriate separator character. For example, you cannot return a Year value if the Day value is omitted. Years must be specified as four digits.

- A time string must be properly formatted with the appropriate separator character. Because a time value can be zero, you can omit one or more time elements (either retaining or omitting the separator characters) and these elements will be returned with a value of zero. Thus, 'hh:mm:ss', 'hh:mm:', 'hh:mm', 'hh::ss', 'hh::', 'hh', and ':::' are all valid. To omit the Hour element, *date-expression* must not have a date portion of the string, and you must retain at least one separator character (:).

- Date and time values must be within a valid range. Years: 0001 through 9999. Months: 1 through 12. Days: 1 through 31. Hours: 0 through 23. Minutes: 0 through 59. Seconds: 0 through 59.

- The number of days in a month must match the month and year. For example, the date '02–29' is only valid if the specified year is a leap year.

- Most date and time values less than 10 may include or omit a leading zero. However, an Hour value of less than 10 must include the leading zero if it is part of a datetime string. Other non-canonical integer values are not permitted. Therefore, a Day value of '07' or '7' is valid, but '007', '7.0' or '7a' are not valid.

- If *date-expression* specifies a time format but does not specify a date format, **DATENAME** does not perform range validation for the time component values.

Arguments

datepart

The type of date/time information to return. The name (or abbreviation) of a date or time part. This name can be specified in uppercase or lowercase, with or without enclosing quotes. The *datepart* can be specified as a literal or a host variable.

date-expression

A date, time, or timestamp expression from which the *datepart* value is to be returned. *date-expression* must contain a value of type *datepart*.

Examples

In the following example, each **DATENAME** returns 'Wednesday' because that is the day of week ('dw') of the specified date:

SQL

```
SELECT DATENAME('dw','2018-02-21') AS DayName,
       DATENAME(dw,'02/21/2018') AS DayName,
       DATENAME('DW',64700) AS DayName
```

The following example returns 'December' because that is the month name ('mm') of the specified date:

SQL

```
SELECT DATENAME('mm','2018-12-20 12:00:00') AS MonthName
```

The following example returns '2018' (as a string) because that is the year ('yy') of the specified date:

SQL

```
SELECT DATENAME('yy','2018-12-20 12:00:00') AS Year
```

Note that the above examples use the abbreviations of the date parts. However, you can specify the full name, as in this example:

SQL

```
SELECT DATENAME('year','2018-12-20 12:00:00') AS Year
```

The following example returns the current quarter, week-of-year, and day-of-year. Each value is returned as a string:

SQL

```
SELECT DATENAME('Q',$HOROLOG) AS Q,
       DATENAME('WK',$HOROLOG) AS WkCnt,
       DATENAME('DY',$HOROLOG) AS DayCnt
```

The following example passes in the *datepart* and *date-expression* as a host variables:

SQL

```
SELECT DATENAME("year",$HOROLOG)
```

The following example uses a subquery to returns records from Sample.Person whose day of birth was a Wednesday:

SQL

```
SELECT Name AS WednesdaysChild,DOB
FROM (SELECT Name,DOB,DATENAME('dw',DOB) AS Wkday FROM Sample.Person)
WHERE Wkday='Wednesday'
ORDER BY DOB
```

See Also

- SQL functions: DATEADD, DATEDIFF, DATEPART, TO_DATE, TIMESTAMPADD, TIMESTAMPDIFF

- ObjectScript function: $ZDATETIME

- **^%DATE** legacy documentation at https://docs.intersystems.com/priordocexcerpts

DATEPART (SQL)

A date/time function that returns an integer representing the value of the specified part of a date/time expression.

Synopsis

```
DATEPART(datepart,date-expression)
```

Arguments

Argument	Description
datepart	The type of date/time information to return. The name (or abbreviation) of a date or time part. This name can be specified in uppercase or lowercase, with or without enclosing quotes. The *datepart* can be specified as a literal or a host variable.
date-expression	A date, time, or timestamp expression from which the *datepart* value is to be returned. *date-expression* must contain a value of type *datepart*.

Description

The **DATEPART** function returns the *datepart* information about a specified date/time expression as data type Integer. The one exception is sqltimestamp (sts), whichreturns as data type %Library.Timestamp. To return *datepart* information as a character string, use DATENAME.

DATEPART returns the value of only one element of *date-expression*; to return a string containing multiple date parts, use TO_DATE.

This function can also be invoked from ObjectScript using the **DATEPART()** method call:

```
$SYSTEM.SQL.Functions.DATEPART(datepart,date-expression)
```

DATEPART is provided for Sybase and Microsoft SQL Server compatibility.

Datepart Argument

The *datepart* argument can be one of the following date/time components, either the full name (the Date Part column) or its abbreviation (the Abbreviation column). These *datepart* component names and abbreviations are not case-sensitive.

Date Part	Abbreviations	Return Values
year	yyyy, yy	0001-9999
quarter	qq, q	1-4
month	mm, m	1-12
week	wk, ww	1-53
weekday	dw	1-7 (Sunday,...,Saturday)
dayofyear	dy, y	1-366
day	dd, d	1-31
hour	hh	0-23
minute	mi, n	0-59

Date Part	Abbreviations	Return Values
second	ss, s	0-59
millisecond	ms	0-999 (with precision of 3)
microsecond	mcs	0–999999 (with precision of 6)
nanosecond	ns	0–999999999 (with precision of 9)
sqltimestamp	sts	SQL_TIMESTAMP: yyyy-mm-dd hh:mm:ss

The preceding table shows the default return values for the various date parts. You can modify the returned values for several of these date parts by using the SET OPTION command with various time and date options.

`week`: InterSystems IRIS can be configured to determine the week of the year for a given date using either the InterSystems IRIS default algorithm or the ISO 8601 standard algorithm. For further details, refer to the WEEK function.

`weekday`: The InterSystems IRIS default for weekday is to designate Sunday as first day of the week (weekday=1). However, you can configure the first day of the week to another value, or you can apply the ISO 8601 standard which designates Monday as first day of the week. For further details, refer to the DAYOFWEEK function. Note that the ObjectScript **$ZDATE** and **$ZDATETIME** functions count week days from 0 through 6 (not 1 through 7).

`second`: If the *date-expression* contains fractional seconds, InterSystems IRIS returns `second` as a decimal number with whole seconds as the integer component, and fractional seconds as the decimal component. Precision is not truncated.

`millisecond`: InterSystems IRIS returns three fractional digits of precision, with trailing zeroes removed. If the *date-expression* has more than three fractional digits of precision, InterSystems IRIS truncates it to three digits.

`sqltimestamp`: InterSystems IRIS converts the input data to timestamp format and supplies zero values for the time elements, if necessary. The `sqltimestamp` (abbreviated `sts`) *datepart* value is for use *only* with **DATEPART**. Do not attempt to use this value in other contexts.

A *datepart* can be specified as a quoted string, without quotes, or with parentheses around a quoted string. No literal substitution is performed on *datepart*, regardless of how specified; literal substitution is performed on *date-expression*. All *datepart* values return a data type INTEGER value, except sqltimestamp (or sts), which returns its value as a character string of data type TIMESTAMP.

Date Input Formats

The *date-expression* argument can be in any of the following formats:

- An InterSystems IRIS %Date logical value (+$H)

- An InterSystems IRIS %PosixTime (%Library.PosixTime) logical value (an encoded 64-bit signed integer)

- An InterSystems IRIS %TimeStamp (%Library.TimeStamp) logical value (YYYY-MM-DD HH:MM:SS.FFF), also known as ODBC format.

- An InterSystems IRIS %String (or compatible) value

The InterSystems IRIS %String (or compatible) value can be in any of the following formats:

- 99999,99999 ($H format)

- *Sybase/SQL-Server-date Sybase/SQL-Server-time*

- *Sybase/SQL-Server-time Sybase/SQL-Server-date*

- *Sybase/SQL-Server-date* (default time is 00:00:00)

- *Sybase/SQL-Server-time* (default date is 01/01/1900)

Sybase/SQL-Server-date is one of these five formats:

```
mmdelimiterdddelimiter[yy]yy dd Mmm[mm][,][yy]yy dd [yy]yy Mmm[mm] yyyy Mmm[mm] dd yyyy
[dd] Mmm[mm]
```

where *delimiter* is a slash (/), hyphen (-), or period (.).

If the year is given as two digits, InterSystems IRIS checks the sliding window to interpret the date. The system default for the sliding window can be set via the **^%DATE** legacy utility. For information on setting the sliding window for the current process, see the ObjectScript $ZDATE, $ZDATEH, $ZDATETIME and $ZDATETIMEH functions.

Sybase/SQL-Server-time represents one of these three formats:

```
HH:MM[:SS:SSS][{AM|PM}] HH:MM[:SS.S] HH[''']{AM|PM}
```

If the *date-expression* specifies a time format but does not specify a date format, **DATENAME** defaults to the date 1900–01–01, which has a weekday value of 2 (Monday).

For sqltimestamp, time is returned as a 24-hour clock. Fractional seconds are truncated.

Invalid Argument Error Codes

If you specify an invalid *datepart* option, **DATEPART** generates an SQLCODE -8 error code, and the following %msg: `'badopt' is not a recognized DATEPART option.`

If you specify an invalid *date-expression* value (for example, an alphabetic text string), **DATEPART** generates an SQLCODE -400 error code, and the following %msg: `Invalid input to DATEPART() function: DATEPART('year','badval').` If you specify a *date-expression* that fails validation (as described below), **DATEPART** generates an SQLCODE -400 error code, and the following %msg: `Unexpected error occurred: <ILLEGAL VALUE>datepart.`

Range and Value Checking

DATEPART performs the following checks on *date-expression* values. If a value fails a check, the null string is returned.

- A valid *date-expression* may consist of a date string (yyyy-mm-dd), a time string (hh:mm:ss), or a date and time string (yyyy-mm-dd hh:mm:ss). If both date and time are specified, both must be valid. For example, you can return a Year value if no time string is specified, but you cannot return a Year value if an invalid time string is specified.

- A date string must be complete and properly formatted with the appropriate number of elements and digits for each element, and the appropriate separator character. For example, you cannot return a Year value if the Day value is omitted. Years must be specified as four digits.

- A time string must be properly formatted with the appropriate separator character. Because a time value can be zero, you can omit one or more time elements (either retaining or omitting the separator characters) and these elements will be returned with a value of zero. Thus, 'hh:mm:ss', 'hh:mm:', 'hh:mm', 'hh::ss', 'hh::', 'hh', and ':::' are all valid. To omit the Hour element, *date-expression* must not have a date portion of the string, and you must retain at least one separator character (:).

- Date and time values must be within a valid range. Years: 0001 through 9999. Months: 1 through 12. Days: 1 through 31. Hours: 0 through 23. Minutes: 0 through 59. Seconds: 0 through 59.

- The number of days in a month must match the month and year. For example, the date '02–29' is only valid if the specified year is a leap year.

- Most date and time values less than 10 may include or omit a leading zero. However, an Hour value of less than 10 must include the leading zero if it is part of a datetime string. Other non-canonical integer values are not permitted. Therefore, a Day value of '07' or '7' is valid, but '007', '7.0' or '7a' are not valid.

- If *date-expression* specifies a time format but does not specify a date format, **DATEPART** does not perform range validation for the time component values.

Examples

In the following example, each **DATEPART** returns the year portion of the datetime string (in this case, 2018) as an integer. Note that *date-expression* can be in various formats, and *datepart* can be specified as either the datepart name or datepart abbreviation, quoted or unquoted:

SQL

```
SELECT DATEPART('yy','2018-02-22 12:00:00') AS YearDTS,
       DATEPART('year','2018-02-22') AS YearDS,
       DATEPART(YYYY,'02/22/2018') AS YearD,
       DATEPART(YEAR,64701) AS YearHD,
       DATEPART('Year','64701,23456') AS YearHDT
```

The following example returns the current year and quarter, based on the $HOROLOG value:

SQL

```
SELECT DATEPART('yyyy',$HOROLOG) AS Year,DATEPART('q',$HOROLOG) AS Quarter
```

The following example returns the birth day-of-week for the Sample.Person table, ordered by day of week:

SQL

```
SELECT Name,DOB,DATEPART('weekday',DOB) AS bday
FROM Sample.Person
ORDER BY bday,DOB
```

In the following example, each **DATEPART** returns 20 as the minutes portion of the *date-expression* string:

SQL

```
SELECT DATEPART('mi','2018-2-20 12:20:07') AS Minutes,
       DATEPART('n','2018-02-20 10:20:') AS Minutes,
       DATEPART(MINUTE,'2018-02-20 10:20') AS Minutes
```

In the following example, each **DATEPART** returns 0 as the seconds portion of the *date-expression* string:

SQL

```
SELECT DATEPART('ss','2018-02-20 03:20:') AS Seconds,
       DATEPART('S','2018-02-20 03:20') AS Seconds,
       DATEPART('Second','2018-02-20') AS Seconds
```

The following example returns the full SQL timestamp as a TIMESTAMP data type. **DATEPART** fills in the missing time information to return a timestamp of '2018-02-25 00:00:00':

SQL

```
SELECT DATEPART(sqltimestamp,'2/25/2018') AS DTStamp
```

The following example supplies a date and time in $HOROLOG format, and returns a timestamp of '2018-02-22 06:30:56':

SQL

```
SELECT DATEPART(sqltimestamp,'64701,23456') AS DTStamp
```

The following example uses a subquery with **DATEPART** to return those people whose birthday is leap year day (February 29th):

SQL

```
SELECT Name,DOB
FROM (SELECT Name,DOB,DATEPART('dd',DOB) AS DayNum,DATEPART('mm',DOB) AS Month FROM Sample.Person)
WHERE Month=2 AND DayNum=29
```

See Also

- **DATEDIFF** function

- **DATENAME** function

- **TIMESTAMPADD** function

- **TIMESTAMPDIFF** function

- **TO_DATE** function

- **^%DATE** legacy documentation at https://docs.intersystems.com/priordocexcerpts

DAY (SQL)

A date function that returns the day of the month for a date expression.

Synopsis

```
DAY(date-expression)
```

```
{fn DAY(date-expression)}
```

Description

DAY returns the day of the month for a given date expression.

The **DAY** function is an alias for the **DAYOFMONTH** function. **DAY** is provided for TSQL compatibility. Refer to DAYOFMONTH for further details.

Arguments

date-expression

An expression that is the name of a column, the result of another scalar function, or a date or timestamp literal.

See Also

- DAYOFMONTH

DAYNAME (SQL)

A date function that returns the name of the day of the week for a date expression.

Synopsis

```
{fn DAYNAME(date-expression)}
```

Description

DAYNAME returns the name of the day that corresponds to a specified date. The returned value is a character string with a maximum length of 15. The default day names returned are: Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday.

To change these default day name values, use the SET OPTION command with the WEEKDAY_NAME option.

The day name is calculated for an InterSystems IRIS date integer, a $HOROLOG or $ZTIMESTAMP value, an ODBC format date string, or a timestamp.

A *date-expression* timestamp can be either data type %Library.PosixTime (an encoded 64-bit signed integer), or data type %Library.TimeStamp (yyyy-mm-dd hh:mm:ss.fff).

The time portion of the timestamp is not evaluated and can be omitted.

DAYNAME checks that the date supplied is a valid date. The year must be between 0001 and 9999 (inclusive), the month 01 through 12, and the day appropriate for that month (for example, 02/29 is only valid on leap years). If the date is not valid, **DAYNAME** issues an SQLCODE -400 error (Fatal error occurred).

The same day of week information can be returned by using the DATENAME function. You can use TO_DATE to retrieve a day name or day name abbreviation with other date elements. To return an integer corresponding to the day of the week, use DAYOFWEEK DATEPART or TO_DATE.

This function can also be invoked from ObjectScript using the **DAYNAME()** method call:

```
$SYSTEM.SQL.Functions.DAYNAME(date-expression)
```

Arguments

date-expression

An expression that evaluates to either an InterSystems IRIS date integer, an ODBC date, or a timestamp. This expression can be the name of a column, the result of another scalar function, or a date or timestamp literal.

Examples

The following examples both return the character string Wednesday because the day of the date (February 21, 2018) is a Wednesday. The first example takes a timestamp string:

SQL

```
SELECT {fn DAYNAME('2018-02-21 12:35:46')} AS Weekday
```

The second example takes an InterSystems IRIS date integer:

SQL

```
SELECT {fn DAYNAME(64700)} AS Weekday
```

The following examples all return the name of the current day of the week:

SQL

```
SELECT {fn DAYNAME({fn NOW()})} AS Wd_Now,
       {fn DAYNAME(CURRENT_DATE)} AS Wd_CurrDate,
       {fn DAYNAME(CURRENT_TIMESTAMP)} AS Wd_CurrTstamp,
       {fn DAYNAME($ZTIMESTAMP)} AS Wd_ZTstamp,
       {fn DAYNAME($HOROLOG)} AS Wd_Horolog
```

Note that **$ZTIMESTAMP** returns Coordinated Universal Time (UTC). The other *time-expression* values return the local time. This may affect the **DAYNAME** value.

The following example shows how **DAYNAME** responds to an invalid date (the year 2017 was not a leap year):

SQL

```
SELECT {fn DAYNAME("2017-02-29")}
```

See Also

- SQL functions: DATENAME, DATEPART, DAYOFMONTH, DAYOFWEEK, DAYOFYEAR, TO_DATE

- ObjectScript function: $ZDATE

- ObjectScript special variables: $HOROLOG, $ZTIMESTAMP

DAYOFMONTH (SQL)

A date function that returns the day of the month for a date expression.

Synopsis

```
{fn DAYOFMONTH(date-expression)}
```

Description

DAYOFMONTH returns the day of the month as an integer from 1 to 31. The *date-expression* can be an InterSystems IRIS date integer, a $HOROLOG or $ZTIMESTAMP value, an ODBC format date string, or a timestamp.

A *date-expression* timestamp can be either data type %Library.PosixTime (an encoded 64-bit signed integer), or data type %Library.TimeStamp (yyyy-mm-dd hh:mm:ss.fff).

The time portion of the timestamp or **$HOROLOG** string is not evaluated and can be omitted.

The **DAYOFMONTH** and **DAY** functions are functionally identical.

This function can also be invoked from ObjectScript using the **DAYOFMONTH()** method call:

ObjectScript

```
WRITE $SYSTEM.SQL.Functions.DAYOFMONTH("2018-02-25")
```

Timestamp date-expression

The day (dd) portion of a timestamp string should be an integer in the range from 1 through 31. There is, however, no range checking for user-supplied values. Numbers greater than 31 and fractions are returned as specified. Because (–) is used as a separator character, negative numbers are not supported. Leading zeros are optional on input; leading zeros are suppressed on output.

DAYOFMONTH returns NULL when the day portion is '0', '00', or a nonnumeric value. NULL is also returned if the day portion of the date string is omitted entirely ('yyyy–mm hh:mm:ss'), or if no date expression is supplied.

The elements of a datetime string can be returned using the following SQL scalar functions: **YEAR**, **MONTH**, **DAYOFMONTH** (or **DAY**), **HOUR**, **MINUTE**, **SECOND**. The same elements can be returned by using the DATEPART or DATENAME function. **DATEPART** and **DATENAME** performs value and range checking on day values.

$HOROLOG date-expression

When calculating day of the month for a **$HOROLOG** value, **DAYOFMONTH** calculates leap years differences, including century day adjustments: 2000 is a leap year, 1900 and 2100 are not leap years.

DAYOFMONTH can process *date-expression* values prior to December 31, 1840 as negative integers. This is shown in the following example:

SQL

```
SELECT {fn DAYOFMONTH(-306)} AS DayOfMonthFeb,      /* February 29, 1840 */
       {fn DAYOFMONTH(-305)} AS DayOfMonthMar,      /* March 1, 1840     */
       {fn DAYOFMONTH(-127410)} AS DayOfMonthFeb    /* February 29, 1492 */
```

The LAST_DAY function returns the date (in **$HOROLOG** format) of the last day of the month for a specified date.

Arguments

date-expression

A date or timestamp expression from which the day of the month value is to be returned. An expression that is the name of a column, the result of another scalar function, or a date or timestamp literal.

Examples

The following examples return the number 25 because the date specified is the twenty-fifth day of the month:

SQL

```
SELECT {fn DAYOFMONTH('2018-02-25')} AS DayNumTS,
       {fn DAYOFMONTH(64704)} AS DayNumH
```

The following example also returns the number 25 for the day of the month. The year is omitted, but the separator character (–) serves as a placeholder:

SQL

```
SELECT {fn DAYOFMONTH('-02-25 11:45:32')} AS DayNum
```

The following examples return <null>:

SQL

```
SELECT{fn DAYOFMONTH('2018-02-00 11:45:32')} AS DayNum
```

SQL

```
SELECT {fn DAYOFMONTH('2018-02 11:45:32')} AS DayNum
```

SQL

```
SELECT {fn DAYOFMONTH('11:45:32')} AS DayNum
```

The following **DAYOFMONTH** examples all returns the current day of the month:

SQL

```
SELECT {fn DAYOFMONTH({fn NOW()})} AS DoM_Now,
       {fn DAYOFMONTH(CURRENT_DATE)} AS DoM_CurrD,
       {fn DAYOFMONTH(CURRENT_TIMESTAMP)} AS DoM_CurrTS,
       {fn DAYOFMONTH($HOROLOG)} AS DoM_Horolog,
       {fn DAYOFMONTH($ZTIMESTAMP)} AS DoM_ZTS
```

Note that **$ZTIMESTAMP** returns Coordinated Universal Time (UTC). The other *time-expression* values return the local time. This may affect the **DAYOFMONTH** value.

The following example shows that leading zeros are suppressed. It returns a length of either 1 or 2, depending on the day of the month value:

SQL

```
SELECT LENGTH({fn DAYOFMONTH('2018-02-05')}),
       LENGTH({fn DAYOFMONTH('2018-02-15')})
```

See Also

- SQL functions: DATENAME, DATEPART, DAY, DAYNAME, DAYOFWEEK, DAYOFYEAR, LAST_DAY, MONTH, TO_DATE

- ObjectScript function: $ZDATE

- ObjectScript special variables: $HOROLOG, $ZTIMESTAMP

DAYOFWEEK (SQL)

A date function that returns the day of the week as an integer for a date expression.

Synopsis

```
{fn DAYOFWEEK(date-expression)}
```

Description

DAYOFWEEK takes a *date-expression* and returns an integer corresponding to the day of the week for that date. Days of the week are counted from the first day of the week; the InterSystems IRIS default is that Sunday is the first day of the week. Therefore, by default, the returned values represent these days:

- 1 — Sunday

- 2 — Monday

- 3 — Tuesday

- 4 — Wednesday

- 5 — Thursday

- 6 — Friday

- 7 — Saturday

The first day of the week default can be overridden system-wide or for specific namespaces, as described in "Setting First Day of Week".

Note that the ObjectScript **$ZDATE** and **$ZDATETIME** functions count days of the week from 0 through 6 (not 1 through 7).

The *date-expression* can be an InterSystems IRIS date integer, a $HOROLOG or $ZTIMESTAMP value, an ODBC format date string, or a timestamp.

A *date-expression* timestamp can be either data type %Library.PosixTime (an encoded 64-bit signed integer), or data type %Library.TimeStamp (yyyy-mm-dd hh:mm:ss.fff).

The time portion of the timestamp is not evaluated and can be omitted.

The same day of week information can be returned by using the DATEPART or TO_DATE function. To return the name of the day of the week, use DAYNAME, DATENAME, or TO_DATE.

This function can also be invoked from ObjectScript using the **DAYOFWEEK()** method call:

```
$SYSTEM.SQL.Functions.DAYOFWEEK(date-expression)
```

Date Validation

DAYOFWEEK performs the following checks on input values. If a value fails a check, the null string is returned.

- A valid *date-expression* may consist of a date string (yyyy-mm-dd), a date and time string (yyyy-mm-dd hh:mm:ss), an InterSystems IRIS date integer, or a $HOROLOG value. **DAYOFWEEK** evaluates only the date portion of the *date-expression*.

- A date string must be complete and properly formatted with the appropriate number of elements and digits for each element, and the appropriate separator character. Years must be specified as four digits.

- Date values must be within a valid range. Years: 0001 through 9999. Months: 1 through 12. Days: 1 through 31.

- The number of days in a month must match the month and year. For example, the date '02–29' is only valid if the specified year is a leap year.

- Date values less than 10 may include or omit a leading zero. Other non-canonical integer values are not permitted. Therefore, a Day value of '07' or '7' is valid, but '007', '7.0' or '7a' are not valid.

Setting First Day of Week

By default, the first day of the week is Sunday. You can override this default system-wide by specifying `SET ^%SYS("sql","sys","day of week")=n`, where *n* values are 1=Monday through 7=Sunday. To set Monday as the first day of the week specify `SET ^%SYS("sql","sys","day of week")=1`. If Monday is the first day of the week, a Wednesday *date-expression* returns 3, rather than the 4 that would be returned if Sunday was the first day of the week. To reset the InterSystems IRIS default (Sunday as first day of week), specify `SET ^%SYS("sql","sys","day of week")=7`.

You can set the first day of the week for a specific namespace by specifying `SET ^%SYS("sql","sys","day of week",namespace)=n`, where *n* values are 1=Monday through 7=Sunday. To set Monday as the first day of the week for the USER namespace, specify `SET ^%SYS("sql","sys","day of week","USER")=1`. Once the first day of the week is set at the namespace level, changing the system-wide setting by specifying `SET ^%SYS("sql","sys","day of week")=n` has no effect on that namespace. To restore the ability to change that namespace's first day of week default, you must kill ^%SYS("sql","sys","day of week",*namespace*). See example below.

InterSystems IRIS also supports the ISO 8601 standard for determining the day of the week, week of the year, and other date settings. This standard is principally used in European countries. The ISO 8601 standard begins counting the days of the week with Monday. To activate ISO 8601, `SET ^%SYS("sql","sys","week ISO8601")=1`; to deactivate, set it to 0. If `week ISO8601` is activated and InterSystems IRIS `day of week` is undefined or set to the default (7=Sunday), the ISO 8601 standard overrides the InterSystems IRIS default. If InterSystems IRIS `day of week` is set to any other value, it overrides `week ISO8601` for **DAYOFWEEK**. See example below.

Arguments

date-expression

A valid ODBC-format date or $HOROLOG format date, with or without the time component. An expression that is the name of a column, the result of another scalar function, or a date or timestamp literal.

Examples

In the following example, both select-items return the number 5 (if Sunday is set as the first day of the week) because the specified *date-expression* (64701 = February 22, 2018) is a Thursday:

SQL

```
SELECT {fn DAYOFWEEK('2018-02-22')}||' '||DATENAME('dw','2018-02-22') AS ODBCDoW,
       {fn DAYOFWEEK(64701)}||' '||DATENAME('dw','64701') AS HorologDoW
```

In the following example, all select-items return the integer corresponding to the current day of the week:

SQL

```
SELECT {fn DAYOFWEEK({fn NOW()})} AS DoW_Now,
       {fn DAYOFWEEK(CURRENT_DATE)} AS DoW_CurrDate,
       {fn DAYOFWEEK(CURRENT_TIMESTAMP)} AS DoW_CurrTstamp,
       {fn DAYOFWEEK($ZTIMESTAMP)} AS DoW_ZTstamp,
       {fn DAYOFWEEK($HOROLOG)} AS DoW_Horolog
```

Note that **$ZTIMESTAMP** returns Coordinated Universal Time (UTC). The other *time-expression* values return the local time. This may affect the **DAYOFWEEK** value.

The following Embedded SQL example demonstrates changing the first day of week for a namespace. It initially sets the system-wide first day of week (to 7), then sets the first day of week for a namespace (to 3). A subsequent system-wide first day of week change (to 2) has no effect on namespace first day of week until the program kills the namespace-specific setting. Killing the namespace-specific setting immediately resets that namespace's first day of week to the current system-wide value. Finally, the program restores the initial system-wide setting.

Note: The following program tests if you have namespace-specific first day of week settings for the %SYS or USER namespaces. If you do, this program aborts to prevent changing these settings.

ObjectScript

```
SetUp
  SET TestNsp="USER"
  SET ControlNsp="%SYS"
InitialDoWValues
  WRITE "Systemwide default DoW initial values",!
  DO TestDayofWeek()
  IF a=b {WRITE "No namespace-specific DoW defaults",!!}
  ELSE {WRITE "DoW initial settings are namespace-specific",!
        WRITE "Stopping this program"
        QUIT }
  SET initialDoW=^%SYS("sql","sys","day of week")
SetSystemwideDoW
  KILL ^%SYS("sql","sys","day of week",TestNsp)
  KILL ^%SYS("sql","sys","day of week",ControlNsp)
  SET ^%SYS("sql","sys","day of week")=7
  WRITE "Systemwide DoW set",!
  DO TestDayofWeek()
SetNamespaceDoW
  SET ^%SYS("sql","sys","day of week",TestNsp)=3
  WRITE TestNsp," namespace DoW set",!
  &sql(SELECT {fn DAYOFWEEK($HOROLOG)} INTO :a)
  DO TestDayofWeek()
ResetSystemwideDoW
  SET ^%SYS("sql","sys","day of week")=2
  WRITE "Systemwide DoW set with ",TestNsp," DoW set",!
  DO TestDayofWeek
KillNamespaceDoW
  KILL ^%SYS("sql","sys","day of week",TestNsp)
  WRITE "Namespace ",TestNsp," DoW killed",!
  DO TestDayofWeek
ResetSystemwideDoWDefault
  SET ^%SYS("sql","sys","day of week")=initialDoW
  WRITE "Systemwide DoW reset after ",TestNsp," DoW killed",!
  DO TestDayofWeek
TestDayofWeek()
  SET $NAMESPACE=TestNsp
  &sql(SELECT {fn DAYOFWEEK($HOROLOG)} INTO :a)
  WRITE "Today is the ",a," day of week in ",$NAMESPACE,!
  SET $NAMESPACE=ControlNsp
  &sql(SELECT {fn DAYOFWEEK($HOROLOG)} INTO :b)
  WRITE "Today is the ",b," day of week in ",$NAMESPACE,!!
  RETURN
```

The following Embedded SQL example shows the default day of the week and the day of the week with the ISO 8601 standard applied. It assumes that the InterSystems IRIS day of week is undefined or set to the default:

ObjectScript

```
TestISO
  SET def=$DATA(^%SYS("sql","sys","week ISO8601"))
  IF def=0 {SET ^%SYS("sql","sys","week ISO8601")=0}
  ELSE {SET isoval=^%SYS("sql","sys","week ISO8601")}
     IF isoval=1 {GOTO UnsetISO }
     ELSE {SET isoval=0 GOTO DayofWeek }
UnsetISO
  SET ^%SYS("sql","sys","week ISO8601")=0
DayofWeek
  &sql(SELECT {fn DAYOFWEEK($HOROLOG)} INTO :a)
  WRITE "Today:",!
  WRITE "default day of week is ",a,!
  SET ^%SYS("sql","sys","week ISO8601")=1
  &sql(SELECT {fn DAYOFWEEK($HOROLOG)} INTO :b)
  WRITE "ISO8601 day of week is ",b,!
ResetISO
  SET ^%SYS("sql","sys","week ISO8601")=isoval
```

See Also

- SQL functions: DATENAME, DATEPART, DAYNAME, DAYOFMONTH, DAYOFYEAR, TO_DATE

- ObjectScript function: $ZDATE

- ObjectScript special variables: $HOROLOG, $ZTIMESTAMP

DAYOFYEAR (SQL)

A date function that returns the day of the year as an integer for a date expression.

Synopsis

```
{fn DAYOFYEAR(date-expression)}
```

Description

DAYOFYEAR returns an integer from 1 to 366 that corresponds to the day of the year for a given date expression. **DAYOFYEAR** calculates leap year dates.

The day of year is calculated for an InterSystems IRIS date integer, a $HOROLOG or $ZTIMESTAMP value, an ODBC format date string, or a timestamp.

A *date-expression* timestamp can be either data type %Library.PosixTime (an encoded 64-bit signed integer), or data type %Library.TimeStamp (yyyy-mm-dd hh:mm:ss.fff).

The time portion of the timestamp is not evaluated and can be omitted.

When calculating day of the month for a **$HOROLOG** value, **DAYOFYEAR** calculates leap years differences, including century day adjustments: 2000 is a leap year, 1900 and 2100 are not leap years.

DAYOFYEAR can process *date-expression* values prior to December 31, 1840 as negative integers. This is shown in the following example:

SQL

```
SELECT {fn DAYOFYEAR(-306)} AS LastDayFeb,  /* February 29, 1840 */
       {fn DAYOFYEAR(-305)} AS FirstDayMar  /* March 1, 1840     */
```

The earliest valid *date-expression* is -672045 (January 1, 0001).

The same day count can be returned by using the DATEPART or DATENAME function. **DATEPART** and **DATENAME** performs value and range checking on date expressions.

This function can also be invoked from ObjectScript using the **DAYOFYEAR()** method call:

```
$SYSTEM.SQL.Functions.DAYOFYEAR(date-expression)
```

Arguments

date-expression

A date expression that is the name of a column, the result of another scalar function, or a date or timestamp literal.

Examples

The following examples both return the number 64 because the day in the date expression (March 4, 2016) is the 64th day of the year (the leap year day is automatically counted):

SQL

```
SELECT {fn DAYOFYEAR('2016-03-04 12:45:37')} AS DayCount
```

SQL

```
SELECT {fn DAYOFYEAR(63981)} AS DayCount
```

The following examples all return the count for the current day:

SQL

```
SELECT {fn DAYOFYEAR({fn NOW()})} AS DNumNow,
       {fn DAYOFYEAR(CURRENT_DATE)} AS DNumCurrD,
       {fn DAYOFYEAR(CURRENT_TIMESTAMP)} AS DNumCurrTS,
       {fn DAYOFYEAR($HOROLOG)} AS DNumHorolog,
       {fn DAYOFYEAR($ZTIMESTAMP)} AS DNumZTS
```

Note that **$ZTIMESTAMP** returns Coordinated Universal Time (UTC). The other *time-expression* values return the local time. This may affect the **DAYOFYEAR** value.

The following example uses a subquery to return Employee records ordered by the day of year of each person's birthday:

SQL

```
SELECT Name,DOB
FROM (SELECT Name,DOB,{fn DAYOFYEAR(DOB)} AS BDay FROM Sample.Employee)
ORDER BY BDay
```

See Also

- SQL functions: DATENAME, DATEPART, DAYNAME, DAYOFMONTH, DAYOFWEEK

- ObjectScript special variables: $HOROLOG, $ZTIMESTAMP

DECODE (SQL)

A function that evaluates a given expression and returns a specified value.

Synopsis

```
DECODE(expr {,search,result}[,default])
```

Description

You can specify multiple *search*,*result* pairs, separated by commas. You can specify one *default*. The maximum number of parameters in the **DECODE** expression (including *expr*, *search*, *result*, and *default*) is about 100. The *search*, *result*, and *default* values can be derived from expressions.

To evaluate a **DECODE** expression, InterSystems IRIS compares *expr* to each *search* value, one by one:

- If *expr* is equal to a *search* value, the corresponding *result* is returned.

- If *expr* is not equal to any *search* value, the *default* value is returned, or, if *default* is omitted, null is returned.

InterSystems IRIS evaluates each *search* value only before comparing it to *expr*, rather than evaluating all *search* values before comparing any of them to *expr*. Therefore, InterSystems IRIS never evaluates a *search* if a previous *search* is equal to *expr*.

In a **DECODE** expression, InterSystems IRIS considers two nulls to be equivalent. If *expr* is null, InterSystems IRIS returns the *result* of the first *search* that is also null.

Note that **DECODE** is supported for Oracle compatibility.

Data Type of Returned Value

DECODE returns the data type of the first *result* argument. If the data type of the first *result* argument cannot be determined, **DECODE** returns VARCHAR. For numeric values, **DECODE** returns the largest length, precision, and scale from all of the possible *result* argument values.

If the data types of *result* and *default* are different, the data type returned is the type most compatible with all of the possible return values, the data type with the highest data type precedence. For example, if *result* is an integer and *default* is a fractional number, **DECODE** returns a value with data type NUMERIC. This is because NUMERIC is the data type with the highest precedence that is compatible with both.

Arguments

expr

The expression to be decoded.

search

The value to which *expr* is compared.

result

The value which is returned if *expr* matches *search*.

default

The optional default value which is returned if *expr* does not match any *search*.

Examples

The following example "decodes" ages from 13 through 19 as 'Teen'; the *default* is 'Adult':

SQL

```
SELECT Name,Age,DECODE(Age,
       13,'Teen',14,'Teen',15,'Teen',16,'Teen',
       17,'Teen',18,'Teen',19,'Teen',
       'Adult') AS AgeBracket
FROM Sample.Person
WHERE Age > 12
```

The following example decodes NULLs. If there is no value for FavoriteColors, **DECODE** replaces it with the string 'No Preference'; otherwise, it returns the FavoriteColors value:

SQL

```
SELECT Name,DECODE(FavoriteColors,
                   NULL,'No Preference',
                   $LISTTOSTRING(FavoriteColors,'^')) AS ColorPreference
FROM Sample.Person
ORDER BY Name
```

The following example decodes color preferences. If the person has a single favorite color, that color name is replaced by a letter abbreviation. If the person has more than one favorite color, **DECODE** returns the FavoriteColors value:

SQL

```
SELECT Name,DECODE(FavoriteColors,
                   $LISTBUILD('Red'),'R',
                   $LISTBUILD('Orange'),'O',
                   $LISTBUILD('Yellow'),'Y',
                   $LISTBUILD('Green'),'G',
                   $LISTBUILD('Blue'),'B',
                   $LISTBUILD('Purple'),'V',
                   $LISTBUILD('White'),'W',
                   $LISTBUILD('Black'),'K',
                   $LISTTOSTRING(FavoriteColors,'^'))
FROM Sample.Person
WHERE FavoriteColors IS NOT NULL
ORDER BY FavoriteColors
```

Note that the ORDER BY clause sorts by the original field values. The following example sorts by the **DECODE** values:

SQL

```
SELECT Name,DECODE(FavoriteColors,
                   $LISTBUILD('Red'),'R',
                   $LISTBUILD('Orange'),'O',
                   $LISTBUILD('Yellow'),'Y',
                   $LISTBUILD('Green'),'G',
                   $LISTBUILD('Blue'),'B',
                   $LISTBUILD('Purple'),'V',
                   $LISTBUILD('White'),'W',
                   $LISTBUILD('Black'),'K',
                   $LISTTOSTRING(FavoriteColors,'^')) AS ColorCode
FROM Sample.Person
WHERE FavoriteColors IS NOT NULL
ORDER BY ColorCode
```

The following example decodes the numeric code in the Company code field in Employee records and returns the corresponding department name. If an employee's Company code is not 1 through 10, DECODE returns the default of "Admin (non-tech)":

SQL

```
SELECT Name,
DECODE (Company,
    1, 'TECH MARKETING', 2, 'TECH SALES', 3, 'DOCUMENTATION',
    4, 'BASIC RESEARCH', 5, 'SOFTWARE DEVELOPMENT', 6, 'HARDWARE DEVELOPMENT',
    7, 'QUALITY TESTING', 8, 'FIELD SUPPORT', 9, 'PHONE SUPPORT',
    10, 'TECH TRAINING',
        'Admin (non-tech)') AS TechJobs
FROM Sample.Employee
```

The expression has Company as the *expr* parameter and uses ten pairs of *search* and *result* parameters. "Admin (non-tech)" is the *default* parameter.

See Also

- CASE

DEGREES (SQL)

A numeric function that converts radians to degrees.

Synopsis

```
DEGREES(numeric-expression)
```

```
{fn DEGREES(numeric-expression)}
```

Description

DEGREES takes an angle measurement in radians and returns the corresponding angle measurement in degrees. **DEGREES** returns NULL if passed a NULL value.

The returned value has a default precision of 36 and a default scale of 18.

You can use the RADIANS function to convert degrees to radians.

Arguments

numeric-expression

The name of the savepoint, specified as an identifier. The measure of an angle in radians. An expression that resolves to a numeric value.

DEGREES returns either the NUMERIC or DOUBLE data type. If *numeric-expression* is data type DOUBLE, **DEGREES** returns DOUBLE; otherwise, it returns NUMERIC.

DEGREES can be specified as either a standard scalar function or an ODBC scalar function with curly brace syntax.

Examples

The following Embedded SQL example returns the degree equivalents corresponding to the radian values 0 through 6:

ObjectScript

```
SET a=0
WHILE a<7 {
&sql(SELECT DEGREES(:a) INTO :b)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE
  QUIT }
ELSE {
  WRITE !,"radians ",a," = degrees ",b
  SET a=a+1 }
}
```

See Also

- SQL functions: CONVERT, RADIANS, TO_NUMBER

%EXACT (SQL)

A collation function that converts characters to the EXACT collation format.

Synopsis

```
%EXACT(expression)
%EXACT expression
```

Description

%EXACT returns *expression* in the EXACT collation sequence. This collation sequence orders values as follows:

1. NULL collates before all actual values. **%EXACT** has no effect on NULLs. This is the same as default collation.

2. Canonical numeric values (whether input as a number or as a string) collate in numeric order before string values.

3. String values collate in case-sensitive string order. The EXACT collation sequence for strings is the same as the ANSI-standard ASCII collation sequence: digits are collated before uppercase alphabetic characters and uppercase alphabetic characters are collated before lowercase alphabetic characters. Punctuation characters occur at several places in the sequence.

This results in a sequence such as the following:

```
NULL
-2          /* canonical number collation */
0
1
2
10
22
88
''          /* empty string */
#           /* character-by-character string collation */
-00         /* non-canonical number collates as string */
0 Elm St.   /* character-by-character string collation */
022         /* non-canonical number collates as string */
1 Elm St.
19 Elm St.
19 elm St. /* string collation is case-sensitive */
19Elm St.
2 Elm St.
201 Elm St.
21 Elm St.
Elm St.
```

%EXACT is commonly used to collate string values containing letters in case-sensitive order. The SQL default is to convert all letters to uppercase for the purpose of collation.

%EXACT is an InterSystems SQL extension and is intended for SQL lookup queries.

You can perform the same collation conversion in ObjectScript using the **Collation**() method of the %SYSTEM.Util class.

%EXACT collates an input string as either wholly numeric (canonical) or as a mixed-character string in which numbers are treated the same as any other character. Compare this to **%MVR** collation, which sorts a string based on the numeric substrings within the string.

DISTINCT and GROUP BY

The DISTINCT clause and the GROUP BY clause group values based on their uppercase default collation, and return values in all uppercase letters, even when none of the actual data values are in all uppercase letters.

*   You can use **%EXACT** to group values by case-sensitive values: SELECT Name FROM mytable GROUP BY %EXACT(Name)

- You can use **%EXACT** to return an actual case-sensitive value for each group: `SELECT %EXACT(Name) FROM mytable GROUP BY Name`

Note: By default, SQL indexes represent string data in uppercase default collation. For this reason, specifying EXACT collation may prevent the use of an index with potentially significant performance implications.

Arguments

commitmode

A string expression, which can be the name of a column, a string literal, a numeric, or the result of another function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR2).

Examples

The following example orders all street addresses by **%EXACT** collation:

SQL

```
SELECT Name,Street
FROM Sample.Person
ORDER BY %EXACT Street
```

The following examples uses **%EXACT** to return all Name values that are higher in the collating sequence than 'Smith'. The first example uses parentheses syntax, the second omits the parentheses.

SQL

```
SELECT Name
FROM Sample.Person
WHERE %EXACT(Name) > 'Smith'
```

SQL

```
SELECT Name
FROM Sample.Person
WHERE %EXACT Name > 'Smith'
```

See Also

- ASCII function

- %SQLSTRING collation function

- %SQLUPPER collation function

- %TRUNCATE collation function

- Collation

EXP (SQL)

A scalar numeric function that returns the exponential (inverse of natural logarithm) of a number.

Synopsis

```
{fn EXP(expression)}
```

Description

EXP is the exponential function e^n, where e is the constant 2.718281828. Therefore, to return the value of e, you can specify {fn EXP(1)}. **EXP** is the inverse of the natural logarithm function LOG.

EXP returns a value with a precision of 36 and a scale of 18. **EXP** returns NULL if passed a NULL value.

EXP can only be used as an ODBC scalar function (with the curly brace syntax).

Arguments

expression

The logarithmic exponent, which is a numeric expression.

EXP returns either the NUMERIC or DOUBLE data type. If *expression* is data type DOUBLE, **EXP** returns DOUBLE; otherwise, it returns NUMERIC.

Examples

The following example returns the constant e:

SQL

```
SELECT {fn EXP(1)} AS e_constant
```

returns 2.718281828...

The following Embedded SQL example returns the exponential values for the integers 0 through 10:

ObjectScript

```
SET a=0
WHILE a<11 {
&sql(SELECT {fn EXP(:a)} INTO :b)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE
  QUIT }
ELSE {
  WRITE !,"Exponential of ",a," = ",b
  SET a=a+1 }
}
```

The following example demonstrates that **EXP** is the inverse of **LOG**:

SQL

```
SELECT {fn EXP(7)} AS Exp,
    {fn LOG(7)} AS Log,
    {fn EXP({fn LOG(7)})} AS ExpOfLog
```

Note in the third function call the small discrepancy between the number input and the calculated return value. The next example shows how to handle this computational discrepancy.

The following Embedded SQL example shows the relationship between the **LOG** and **EXP** functions for the integers 1 through 10:

ObjectScript

```
SET a=1
WHILE a<11 {
&sql(SELECT {fn LOG(:a)} INTO :b)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE
  QUIT }
ELSE {
  WRITE !,"Logarithm of ",a," = ",b }
&sql(SELECT ROUND({fn EXP(:b)},12) INTO :c)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"Exponential of log ",b," = ",c
SET a=a+1 }
}
```

Note that the **ROUND** function is needed here to correct for very small discrepancies caused by system calculation limitations. In the above example, **ROUND** is set arbitrarily to 12 decimal digits for this purpose.

See Also

- SQL functions: LOG LOG10 POWER ROUND

- ObjectScript function: $ZEXP

%EXTERNAL (SQL)

A format-transformation function that returns an expression in DISPLAY format.

Synopsis

```
%EXTERNAL(expression)
%EXTERNAL expression
```

Description

%EXTERNAL converts *expression* to DISPLAY format, regardless of the current select mode (display mode). The DISPLAY format represents data in the VARCHAR data type with whatever data conversion the field or data type Logical-ToDisplay method performs.

%EXTERNAL is commonly used on a **SELECT** list *select-item*. It can be used in a **WHERE** clause, but this use is discouraged because using **%EXTERNAL** prevents the use of indexes on the specified field.

Applying **%EXTERNAL** changes the column header name to a value such as "Expression_1"; it is therefore usually desirable to specify a column name alias, as shown in the examples below.

Whether **%EXTERNAL** converts a date depends on the data type returned by the date field or function. **%EXTERNAL** converts CURDATE, CURRENT_DATE, CURTIME, and CURRENT_TIME values. It does not convert CURRENT_TIMESTAMP, GETDATE, GETUTCDATE, NOW, and **$HOROLOG** values.

When **%EXTERNAL** converts a %List structure to DISPLAY format, the displayed list elements appear to be separated by a blank space. This "space" is actually the two non-display characters CHAR(13) and CHAR(10).

%EXTERNAL is an InterSystems SQL extension.

To convert an *expression* to LOGICAL format, regardless of the current select mode, use the %INTERNAL function. To convert an *expression* to ODBC format, regardless of the current select mode, use the %ODBCOUT function.

For further details on display format options, refer to Data Display Options.

Arguments

expression

The expression to be converted. A field name, an expression containing a field name, or a function that returns a value in a convertible data type, such as DATE or %List. Cannot be a stream field.

Examples

The following Dynamic SQL example returns Date of Birth (DOB) data values in the current select mode format, and the same data using the **%EXTERNAL** function. For the purpose of demonstration, in this program the %SelectMode value is determined randomly for each invocation:

SQL

```
SELECT TOP 5 DOB,%EXTERNAL(DOB) AS ExtDOB
FROM Sample.Person
```

The following examples show the two syntax forms for this function; they are otherwise identical. They specify the **%EXTERNAL** (DISPLAY format), **%INTERNAL** (LOGICAL format), and **%ODBCOUT** (ODBC format) of a %List field:

SQL

```
SELECT TOP 10 %EXTERNAL(FavoriteColors) AS ExtColors,
              %INTERNAL(FavoriteColors) AS IntColors,
              %ODBCOUT(FavoriteColors) AS ODBCColors
FROM Sample.Person
```

SQL

```
SELECT TOP 10 %EXTERNAL FavoriteColors AS ExtColors,
              %INTERNAL FavoriteColors AS IntColors,
              %ODBCOUT FavoriteColors AS ODBCColors
FROM Sample.Person
```

The following example converts date of birth (DOB) and rounded date of birth (DOB) values to **%EXTERNAL** (DISPLAY format):

SQL

```
SELECT %EXTERNAL(DOB) AS DOB,
       %INTERNAL(ROUND(DOB,-3)) AS DOBGroup,
       %EXTERNAL(ROUND(DOB,-3)) AS RoundedDOB
FROM Sample.Person
GROUP BY (ROUND(DOB,-3))
ORDER BY DOBGroup
```

See Also

- %INTERNAL, %ODBCIN, %ODBCOUT

- SQL concepts: Data Types, Date and Time Constructs

$EXTRACT (SQL)

A string function that extracts characters from a string by position.

Synopsis

```
$EXTRACT(string[,from[,to]])
```

Description

$EXTRACT returns a substring from a specified position in *string*. The nature of the substring returned depends on the arguments used.

- **$EXTRACT**(*string*) extracts the first character in the string.

- **$EXTRACT**(*string*,*from*) extracts the character in the position specified by *from*. For example, if variable *var1* contains the string "ABCD", the following command extracts "B" (the second character):

  SQL

  ```
  SELECT $EXTRACT('ABCD',2) AS Extracted
  ```

- **$EXTRACT**(*string*,*from*,*to*) extracts the range of characters starting with the *from* position and ending with the *to* position. For example, the following command extracts the string "Alabama" (that is, all characters from position 5 to position 11, inclusive) from the string "1234Alabama567":

  SQL

  ```
  SELECT $EXTRACT('1234Alabama567',5,11) AS Extracted
  ```

This function returns data of type VARCHAR.

Arguments

string

The *string* value can be a variable name, a numeric value, a string literal, or any valid expression.

from

The *from* value must be a positive integer (however, see Note). If a fractional number, the fraction is truncated and only the integer portion is used.

If the *from* value is greater than the number of characters in the string, **$EXTRACT** returns a null string.

If *from* is specified without the *to* argument, it extracts the single specified character.

If used with the *to* argument, it identifies the start of the range to be extracted and must be less than the value of *to*. If *from* equals *to*, **$EXTRACT** returns the single character at the specified position. If *from* is greater than *to*, **$EXTRACT** returns a null string.

to

The *to* argument must be used with the *from* argument. It must be a positive integer. If a fractional number, the fraction is truncated and only the integer portion is used.

If the *to* value is greater than or equal to the *from* value, **$EXTRACT** returns the specified substring. If *to* is greater than the length of the string, **$EXTRACT** returns the substring from the *from* position to the end of the string. If *to* is less than *from*, **$EXTRACT** returns a null string.

Examples

The following example returns "S", the fourth character in the string:

SQL

```
SELECT $EXTRACT('THIS IS A TEST',4) AS Extracted
```

The following example returns a substring "THIS IS" which is composed of the first through seventh characters.

SQL

```
SELECT $EXTRACT('THIS IS A TEST',1,7) AS Extracted
```

The following example extracts the second character ("B") from "ABCD".

SQL

```
SELECT $EXTRACT("ABCD",2)
```

The following example shows that the one-argument format is equivalent to the two-argument format when the *from* value is "1". Both **$EXTRACT** functions return "H".

SQL

```
SELECT $EXTRACT("HELLO")
SELECT $EXTRACT("HELLO",1)
```

Notes

$EXTRACT Compared with $PIECE and $LIST

$EXTRACT returns a substring of characters by integer position from a string. **$PIECE** and **$LIST** both work on specially formatted strings.

$PIECE returns a substring from a standard character string using delimiter characters within the string.

$LIST returns a sublist of elements from an encoded list by the integer position of elements (not characters). **$LIST** cannot be used on ordinary strings, and **$EXTRACT** cannot be used on encoded lists.

The **$EXTRACT**, **$FIND**, **$LENGTH**, and **$PIECE** functions operate on standard character strings. The various **$LIST** functions operate on encoded character strings, which are incompatible with standard character strings. The only exceptions are the **$LISTGET** function and the one-argument and two-argument forms of **$LIST**, which take an encoded character string as input, but output a single element value as a standard character string.

$EXTRACT and Unicode

The **$EXTRACT** function operates on characters, not bytes. Therefore, Unicode strings are handled the same as ASCII strings, as shown in the following embedded SQL example using the Unicode character for "pi" ($CHAR(960)):

ObjectScript

```
SET a="QT PIE"
SET b=("QT "_$CHAR(960))
&sql(SELECT
$EXTRACT(:a,-33,4),
$EXTRACT(:a,4,4),
$EXTRACT(:a,4,99),
$EXTRACT(:b,-33,4),
$EXTRACT(:b,4,4),
$EXTRACT(:b,4,99)
INTO :a1,:a2,:a3,:b1,:b2,:b3)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"ASCII form returns ",!,a1,!,a2,!,a3
  WRITE !,"Unicode form returns ",!,b1,!,b2,!,b3 }
```

Null and Invalid Arguments

- When *string* is a null string, a null string is returned.

- When *from* is a number larger than the string length, a null string is returned.

- When *from* is zero or a negative number, and no *to* is specified, a null string is returned.

- When *to* is zero, a negative number, or a number smaller than *from*, a null string is returned.

- When *to* is a valid value, *from* can be zero or a negative number. **$EXTRACT** treats such *from* values as 1.

No SQLCODE error is generated for invalid argument values.

In following example, the negative *from* value is evaluated as 1; **$EXTRACT** returns the substring "THIS IS" composed of the first through seventh characters.

SQL

```
SELECT $EXTRACT('THIS IS A TEST',-7,7)
```

In following embedded SQL example, all **$EXTRACT** function calls return the null string:

ObjectScript

```
SET a="THIS IS A TEST"
SET b=""
&sql(SELECT
$EXTRACT(:a,33),
$EXTRACT(:a,-7),
$EXTRACT(:a,3,2),
$EXTRACT(:a,-7,0),
$EXTRACT(:a,-7,-10),
$EXTRACT(:b,-33,4),
$EXTRACT(:b,4,4),
$EXTRACT(:b,4,99),
$EXTRACT(NULL,-33,4),
$EXTRACT(NULL,4,4),
$EXTRACT(NULL,4,99)
INTO :a1,:a2,:a3,:a4,:a5,:b1,:b2,:b3,:c1,:c2,:c3)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"FROM too big: ",a1
  WRITE !,"FROM negative, no TO: ",a2
  WRITE !,"TO smaller than FROM: ",a3
  WRITE !,"TO not a positive integer: ",a4,a5
  WRITE !,"LIST is null string: ",b1,b2,b3,c1,c2,c3 }
```

See Also

- SQL functions: $FIND $LENGTH $LIST $LISTGET $PIECE

- ObjectScript functions: $EXTRACT $FIND $LENGTH $LIST $LISTBUILD $LISTGET $PIECE

$FIND (SQL)

A string function that returns the end position of a substring within a string, with optional search start point.

Synopsis

```
$FIND(string,substring[,start])
```

Description

$FIND returns an integer specifying the end position of a substring within a string. **$FIND** searches *string* for *substring*. If *substring* is found, **$FIND** returns the integer position of the first character following *substring*. If *substring* is not found, **$FIND** returns a value of 0.

You can include the *start* option to specify a starting position for the search. If *start* is greater than the number of characters in *string*, **$FIND** returns a value of 0. If *start* is omitted, string position 1 is the default. If *start* is zero, a negative number, or a nonnumeric string, position 1 is the default.

$FIND is case-sensitive. Use one of the case-conversion functions to locate both uppercase and lowercase instances of a letter or character string.

$FIND, POSITION, CHARINDEX, and INSTR

$FIND, **POSITION**, **CHARINDEX**, and **INSTR** all search a string for a specified substring and return an integer position corresponding to the first match. **$FIND** returns the integer position of the first character after the end of the matching substring. **CHARINDEX**, **POSITION**, and **INSTR** return the integer position of the first character of the matching substring. **CHARINDEX**, **$FIND**, and **INSTR** support specifying a starting point for substring search. **INSTR** also support specifying the substring occurrence from that starting point.

The following example demonstrates these four functions, specifying all optional arguments. Note that the positions of *string* and *substring* differ in these functions:

SQL

```
SELECT POSITION('br' IN 'The broken brown briefcase') AS Position,
       CHARINDEX('br','The broken brown briefcase',6) AS Charindex,
       $FIND('The broken brown briefcase','br',6) AS Find,
       INSTR('The broken brown briefcase','br',6,2) AS Inst
```

For a list of functions that search for a substring, refer to String Manipulation.

Arguments

string

The target string that is to be searched. It can be a variable name, a numeric value, a string literal, or any valid expression.

substring

The substring that is to be searched for. It can be a variable name, a numeric value, a string literal, or any valid expression.

start

An optional argument that denotes the starting point for substring search, specified as a positive integer. A character count from the beginning of *string*, counting from 1. To search from the beginning of *string*, omit this argument or specify a *start* of 0 or 1. A negative number, the empty string, or a nonnumeric value is treated as 0. Specifying *start* as NULL causes **$FIND** to return <null>.

$FIND returns the SMALLINT data type.

Examples

In the following example, *string* contains the string "ABCDEFG" and *substring* contains the string "BCD". The **$FIND** function returns the value 5, indicating the position of the character ("E") that follows "BCD":

SQL

```
SELECT $FIND('ABCDEG','BCD') AS SubPoint
```

The following example searches the numeric 987654321 for the number 7. It returns 4, the position following the *substring*:

SQL

```
SELECT $FIND(987654321,7) AS SubPoint
```

The following example returns 3, the position of the character that follow the first instance of the *substring* "AA":

SQL

```
SELECT $FIND('AAAAAA','AA') AS SubPoint
```

In the following example, **$FIND** searches for a substring that is not in the string. It returns zero (0):

SQL

```
SELECT $FIND('AABBCCDD','AC') AS SubPoint
```

In the following example, **$FIND** begins its search with the seventh character. This example returns 14, the position of the character that follows the next occurrence of "R":

SQL

```
SELECT $FIND('EVERGREEN FOREST','R',7) AS SubPoint
```

In the following example, **$FIND** begins its search after the last character in string. It returns zero (0):

SQL

```
SELECT $FIND('ABCDEFG','G',10) AS SubPoint
```

The following example shows that a *start* less than 1 is treated as 1:

SQL

```
SELECT
$FIND("ABCDEFG","F"),
$FIND("ABCDEFG","F",1),
$FIND("ABCDEFG","F",0),
$FIND("ABCDEFG","F",-35)
```

The following example uses **$FIND** to search a string containing the Unicode character for pi, $CHAR(960). The first **$FIND** returns 5, the character following pi. The second **$FIND** also returns 5; it begins its search at character 4, which happens to be pi, the character sought. The third **$FIND** begins its search at character 5; it returns 13, the position following the next occurrence of pi. Note that position 13 is returned, even though position 12 is the last character in the string:

ObjectScript

```
SELECT
$FIND("QT "_$CHAR(960)_" HONEY "_$CHAR(960),$CHAR(960)),
$FIND("QT "_$CHAR(960)_" HONEY "_$CHAR(960),$CHAR(960),4),
$FIND("QT "_$CHAR(960)_" HONEY "_$CHAR(960),$CHAR(960),5)
```

See Also

- CHARINDEX function
- INSTR function
- POSITION function
- String Manipulation

FLOOR (SQL)

A numeric function that returns the largest integer less than or equal to a given numeric expression.

Synopsis

```
FLOOR(numeric-expression)
{fn FLOOR(numeric-expression)}
```

Description

FLOOR returns the nearest integer value less than or equal to *numeric-expression*. The returned value has a scale of 0. When *numeric-expression* is a NULL value, an empty string ("), or a nonnumeric string, **FLOOR** returns NULL.

Note that **FLOOR** can be invoked as an ODBC scalar function (with the curly brace syntax) or as an SQL general function.

This function can also be invoked from ObjectScript using the **FLOOR()** method call:

```
$SYSTEM.SQL.Functions.FLOOR(numeric-expression)
```

Arguments

numeric-expression

A number whose floor is to be calculated.

Examples

The following examples show how **FLOOR** converts a fraction to its floor integer:

SQL

```
SELECT FLOOR(167.111) AS FloorNum1,
       FLOOR(167.456) AS FloorNum2,
       FLOOR(167.999) AS FloorNum3
```

all return 167.

SQL

```
SELECT {fn FLOOR(167.00)} AS FloorNum1,
       {fn FLOOR(167)} AS FloorNum2
```

return 167.

SQL

```
SELECT FLOOR(-167.111) AS FloorNum1,
       FLOOR(-167.456) AS FloorNum2,
       FLOOR(-167.999) AS FloorNum3
```

all return -168.

SQL

```
SELECT FLOOR(-167.00) AS FloorNum
```

returns -167.

The following example uses a subquery to reduce a large table of US Zip Codes (postal codes) to one representative city for each floor Latitude integer:

SQL

```
SELECT City,State,FLOOR(Latitude) AS FloorLatitude
FROM (SELECT City,State,Latitude,FLOOR(Latitude) AS FloorNum
      FROM Sample.USZipCode)
GROUP BY FloorNum
ORDER BY FloorNum DESC
```

See Also

- CEILING

- ROUND

GETDATE (SQL)

A date/time function that returns the current local date and time.

Synopsis

```
GETDATE([precision])
```

Arguments

Argument	Description
precision	*Optional* — A positive integer that specifies the time precision as the number of digits of fractional seconds. The default is 0 (no fractional seconds); this default is configurable. A *precision* value is optional, the parentheses are mandatory.

Description

GETDATE returns the current local date and time for this timezone as a timestamp; it adjusts for local time variants, such as Daylight Saving Time.

GETDATE can return a timestamp in either %TimeStamp data type format (yyyy-mm-dd hh:mm:ss.ffff) or %PosixTime data type format (an encoded 64-bit signed integer). The following rules determine which timestamp format is returned:

1. If the current timestamp is being supplied to a field of data type %PosixTime, the current timestamp value is returned in POSIXTIME data type format. For example, `WHERE PosixField=GETDATE()` or `INSERT INTO MyTable (PosixField) VALUES (GETDATE())`.

2. If the current timestamp is being supplied to a field of data type %TimeStamp, the current timestamp value is returned in TIMESTAMP data type format. Its ODBC type is TIMESTAMP, LENGTH is 16, and PRECISION is 19. For example, `WHERE TSField=GETDATE()` or `INSERT INTO MyTable (TSField) VALUES (GETDATE())`.

3. If the current timestamp is being supplied without context, the current timestamp value is returned in TIMESTAMP data type format. For example, `SELECT GETDATE()`.

To change the default datetime string format, use the SET OPTION command with the various date and time options.

GETDATE is a synonym for CURRENT_TIMESTAMP and is provided for compatibility with Sybase and Microsoft SQL Server. The CURRENT_TIMESTAMP and NOW functions can also be used to return the current local date and time as a timestamp in either TIMESTAMP or POSIXTIME formats. **CURRENT_TIMESTAMP** supports precision, **NOW** does not support precision.

To return just the current date, use CURDATE or CURRENT_DATE. To return just the current time, use CURRENT_TIME or CURTIME. These functions use the DATE or TIME data type. None of these functions support precision. The TIME and DATE data types store their values as integers in $HOROLOG format. They can be displayed in either Display format or Logical (storage) format. You can use the CAST or CONVERT function to change the data type of dates and times.

Universal Time (UTC)

All InterSystems SQL timestamp, date, and time functions except GETUTCDATE are specific to the local time zone setting. GETUTCDATE returns the current UTC (universal) date and time as either a TIMESTAMP value or a POSIXTIME value. You can also use the ObjectScript $ZTIMESTAMP special variable to get a current timestamp that is universal (independent of time zone).

Fractional Seconds Precision

GETDATE can return up to nine digits of precision. The number of digits of precision returned is set using the *precision* argument. The default for the *precision* argument can be configured using the following:

- SET OPTION with the TIME_PRECISION option.

- The system-wide **$SYSTEM.SQL.Util.SetOption()** method configuration option `DefaultTimePrecision`. To determine the current setting, call **$SYSTEM.SQL.CurrentSettings()** which displays `Default time precision`; the default is 0.

- Go to the Management Portal, select **System Administration**, **Configuration**, **SQL and Object Settings**, **SQL**. View and edit the current setting of **Default time precision for GETDATE(), CURRENT_TIME, and CURRENT_TIMESTAMP**.

Specify an integer 0 through 9 (inclusive) for the *precision* argument. The default is 0. The actual precision returned is platform dependent; *precision* digits in excess of the precision available on your system are returned as zeroes.

Fractional seconds are always truncated, not rounded, to the specified precision.

Examples

In designing a report, **GETDATE** can be used to print the current date and time each time the report is produced. **GETDATE** is also useful for tracking activity, such as logging the time that a transaction occurred.

GETDATE can be used in a **SELECT** statement select list or in the **WHERE** clause of a query. The following example returns the current date and time in TIMESTAMP format:

SQL

```
SELECT GETDATE() AS DateTime
```

The following example returns the current date and time with two digits of precision:

SQL

```
SELECT GETDATE(2) AS DateTime
```

The following example compares local (time zone specific) and universal (time zone independent) timestamps:

ObjectScript

```
SELECT GETDATE(), GETUTCDATE()
```

The following example sets the LastUpdate field in the selected row of the Orders table to the current system date and time. If LastUpdate is data type %TimeStamp, **GETDATE** returns the current date and time as an ODBC timestamp; if LastUpdate is data type %PosixTime, **GETDATE** returns the current date and time as an encoded 64-bit signed integer:

SQL

```
UPDATE Orders SET LastUpdate = GETDATE()
  WHERE Orders.OrderNumber=:ord
```

GETDATE can be used in **CREATE TABLE** to specify the default value for a given field. In the following example, the **CREATE TABLE** statement uses **GETDATE** to set a default value for the StartDate field:

SQL

```
CREATE TABLE Employees(
    EmpId       INT NOT NULL,
    LastName    CHAR(40) NOT NULL,
    FirstName   CHAR(20) NOT NULL,
    StartDate   TIMESTAMP DEFAULT GETDATE())
```

See Also

- SQL concepts: Data Type, Date and Time Constructs

- SQL timestamp functions: CAST, CONVERT, CURRENT_TIMESTAMP, GETUTCDATE, NOW, SYSDATE, TIMESTAMPADD, TIMESTAMPDIFF, TO_POSIXTIME, TO_TIMESTAMP

- SQL current date and time functions: CURDATE, CURRENT_DATE, CURRENT_TIME, CURTIME

- ObjectScript: $ZDATETIME function, $HOROLOG special variable, $ZTIMESTAMP special variable

GETUTCDATE (SQL)

A date/time function that returns the current UTC date and time.

Synopsis

```
GETUTCDATE([precision])
```

Description

GETUTCDATE returns Universal Time Constant (UTC) date and time as a timestamp. Because UTC time is the same everywhere on the planet, does not depend on the local timezone and is not subject to local time variants (such as Daylight Saving Time), this function is useful for applying consistent timestamps when users in different time zones access the same database.

GETUTCDATE can return a timestamp in either %TimeStamp data type format (yyyy-mm-dd hh:mm:ss.ffff) or %PosixTime data type format (an encoded 64-bit signed integer). The following rules determine which timestamp format is returned:

1. If the current UTC timestamp is being supplied to a field of data type %PosixTime, this timestamp value is returned in POSIXTIME data type format. For example, `WHERE PosixField=GETUTCDATE()` or `INSERT INTO MyTable (PosixField) VALUES (GETUTCDATE())`.

2. If the current UTC timestamp is being supplied to a field of data type %TimeStamp, this timestamp value is returned in TIMESTAMP data type format. Its ODBC type is TIMESTAMP, LENGTH is 16, and PRECISION is 19. For example, `WHERE TSField=GETUTCDATE()` or `INSERT INTO MyTable (TSField) VALUES (GETUTCDATE())`.

3. If the current UTC timestamp is being supplied without context, this timestamp value is returned in TIMESTAMP data type format. For example, `SELECT GETUTCDATE()`.

To change the default datetime string format, use the SET OPTION command with the various date and time options.

Typical uses for **GETUTCDATE** are in the **SELECT** statement select list or in the **WHERE** clause of a query. In designing a report, **GETUTCDATE** can be used to print the current date and time each time the report is produced. **GETUTCDATE** is also useful for tracking activity, such as logging the time that a transaction occurred.

GETUTCDATE can be used in **CREATE TABLE** to specify a field's default value.

Other SQL Functions

GETUTCDATE returns the current UTC date and time as a timestamp in either TIMESTAMP or POSIXTIME format.

All other timestamp functions return the local date and time: GETDATE, CURRENT_TIMESTAMP, NOW, and SYSDATE return the current local date and time as a timestamp in either TIMESTAMP or POSIXTIME format.

GETDATE and **CURRENT_TIMESTAMP** provide a *precision* argument.

NOW, argumentless **CURRENT_TIMESTAMP**, and **SYSDATE** do not provide a *precision* argument; they take the system-wide default time precision.

CURDATE and CURRENT_DATE return the current local date. CURTIME and CURRENT_TIME return the current local time. These functions use the DATE or TIME data type. None of these functions support precision.

A TIMESTAMP data type stores and displays its value in the same format. A POSIXTIME data type stores its value as an encoded 64-bit signed integer. The TIME and DATE data types store their values as integers in $HOROLOG format and can be displayed in a variety of formats.

Note that all InterSystems SQL timestamp functions except **GETUTCDATE** are specific to the local time zone setting. To get a current timestamp that is universal (independent of time zone) you can also use the ObjectScript $ZTIMESTAMP special variable. Note that you can set the *precision* for **GETUTCDATE**; **$ZTIMESTAMP** always returns a precision of 3.

Fractional Seconds Precision

GETUTCDATE can return up to nine digits of precision. The number of digits of precision returned is set using the *precision* argument. The default for the *precision* argument can be configured using the following:

- SET OPTION with the TIME_PRECISION option.

- The system-wide **$SYSTEM.SQL.Util.SetOption()** method configuration option `DefaultTimePrecision`. To determine the current setting, call **$SYSTEM.SQL.CurrentSettings()** which displays `Default time precision`; the default is 0.

- Go to the Management Portal, select **System Administration**, **Configuration, SQL and Object Settings**, **SQL**. View and edit the current setting of **Default time precision for GETDATE(), CURRENT_TIME, and CURRENT_TIMESTAMP**.

Specify an integer 0 through 9 (inclusive) for the default number of decimal digits of precision to return. The default is 0. The actual precision returned is platform dependent; *precision* digits in excess of the precision available on your system are returned as zeroes.

Fractional seconds are always truncated, not rounded, to the specified precision.

Arguments

precision

An optional positive integer that specifies the time precision as the number of digits of fractional seconds. The default is 0 (no fractional seconds); this default is configurable.

Examples

The following example returns the current date and time as a UTC timestamp and as a local timestamp, both in TIMESTAMP format:

SQL

```
SELECT GETUTCDATE() AS UTCDateTime,
       GETDATE() AS LocalDateTime
```

The following example returns the current UTC date and time with fractional seconds having two digits of precision:

SQL

```
SELECT GETUTCDATE(2) AS DateTime
```

The following example compares local (time zone specific) and universal (time zone independent) timestamps:

SQL

```
SELECT GETDATE(),GETUTCDATE()
```

The following example sets the LastUpdate field in the selected row of the Orders table to the current UTC date and time. If LastUpdate is data type %TimeStamp, **GETUTCDATE** returns the current UTC date and time as an ODBC timestamp; if LastUpdate is data type %PosixTime, **GETUTCDATE** returns the current UTC date and time as an encoded 64-bit signed integer:

SQL

```
UPDATE Orders SET LastUpdate = GETUTCDATE()
  WHERE Orders.OrderNumber=:ord
```

In the following example, the **CREATE TABLE** statement uses **GETUTCDATE** to set a default value for the OrderRcvd field:

```
CREATE TABLE Orders(
     OrderId     INT NOT NULL,
     ItemName    CHAR(40) NOT NULL,
     Quantity    INT NOT NULL,
     OrderRcvd   TIMESTAMP DEFAULT GETUTCDATE())
```

See Also

- SQL concepts: Data Type, Date and Time Constructs

- SQL timestamp functions: CAST, CONVERT, CURRENT_TIMESTAMP, GETDATE, NOW, SYSDATE, TIMESTAMPADD, TIMESTAMPDIFF, TO_POSIXTIME, TO_TIMESTAMP

- SQL current date and time functions: CURDATE, CURRENT_DATE, CURRENT_TIME, CURTIME

- ObjectScript: $ZDATETIME function, $HOROLOG special variable, $ZTIMESTAMP special variable

GREATEST (SQL)

A function that returns the greatest value from a series of expressions.

Synopsis

```
GREATEST(expression,expression[,...])
```

Description

GREATEST returns the greatest value from a comma-separated series of expressions. Expressions are evaluated in left-to-right order. If only one *expression* is provided, **GREATEST** returns that value. If any *expression* is NULL, **GREATEST** returns NULL.

If all of the *expression* values resolve to canonical numbers, they are compared in numeric order. If a quoted string contains a number in canonical format, it is compared in numeric order. However, if a quoted string contains a number not in canonical format (for example, '00', '0.4', or '+4'), it is compared as a string. String comparisons are performed character-by-character in collation order. Any string value is greater than any numeric value.

The empty string is greater than any numeric value, but less than any other string value.

If the returned value is a number, **GREATEST** returns it in canonical format (leading and trailing zeros removed, etc.). If the returned value is a string, **GREATEST** returns it unchanged, including any leading or trailing blanks.

GREATEST returns the greatest value from a comma-separated series of expressions. LEAST returns the least value from a comma-separated series of expressions. COALESCE returns the first non-NULL value from a comma-separated series of expressions.

Data Type of Returned Value

If the data types of the *expression* values are different, the data type returned is the type most compatible with all of the possible return values, the data type with the highest data type precedence. For example, if one *expression* is an integer and another *expression* is a fractional number, **GREATEST** returns a value with data type NUMERIC. This is because NUMERIC is the data type with the highest precedence that is compatible with both. If, however, an *expression* is a literal number or string, **GREATEST** returns data type VARCHAR.

Arguments

expression

An expression that resolves to a number or a string. The values of these expressions are compared to each other. An *expression* can be a field name, a literal, an arithmetic expression, a host variable, or an object reference. You can list up to 140 comma-separated expressions.

Examples

In the following example, each **GREATEST** compares three canonical numbers:

SQL

```
SELECT GREATEST(22,2.2,-21) AS HighNum,
       GREATEST('2.2','22','-21') AS HighNumStr
```

In the following example, each **GREATEST** compares three numeric strings. However, each **GREATEST** contains one string that is non-canonical; these non-canonical values are compared as character strings. A character string is always greater than a number:

SQL

```
SELECT GREATEST('22','+2.2','-21'),
        GREATEST('0.2','22','-21')
```

In the following example, each **GREATEST** compare three strings and returns the value with the highest collation sequence:

SQL

```
SELECT GREATEST('A','a',''),
        GREATEST('a','ab','abc'),
        GREATEST('#','0','7'),
        GREATEST('##','00','77')
```

The following example compares two dates, treated as canonical numbers: the date of birth as a $HOROLOG integer, and the integer 58073 converted to a date. It returns the date of birth for each person born in the 21st century. Anyone born before January 1, 2000 is displayed with the default birth date of December 31, 1999:

SQL

```
SELECT Name,GREATEST(DOB,TO_DATE(58073)) AS NewMillenium
FROM Sample.Person
```

See Also

- LEAST function

- COALESCE function

- CONVERT function

- TO_NUMBER function

HOUR (SQL)

A time function that returns the hour for a datetime expression.

Synopsis

```
{fn HOUR(time-expression)}
```

Description

HOUR returns an integer specifying the hour for a given time or datetime value. The hour is calculated for a $HOROLOG or $ZTIMESTAMP value, an ODBC format date string, or a timestamp.

A *time-expression* timestamp can be either data type %Library.PosixTime (an encoded 64-bit signed integer), or data type %Library.TimeStamp (yyyy-mm-dd hh:mm:ss.fff).

To change this default time format, use the SET OPTION command.

Note that you can supply a time integer (number of elapsed seconds), but not a time string (hh:mm:ss). You must supply a datetime string (yyyy-mm-dd hh:mm:ss). You can omit the seconds (:ss) or minutes and seconds (mm:ss) portion of a datetime string and still return the hour portion. The time portion of a datetime string must be a valid time value. The date portion of the datetime string is not validated.

Hours are expressed in 24-hour time. The hours (hh) portion should be an integer in the range from 0 through 23. Leading zeros are optional on input; leading zeros are suppressed on output.

HOUR returns a value of 0 hours when the hours portion is '0' or '00'. Zero hours is also returned if no time expression is supplied, or if the hours portion of the time expression is omitted (':mm:ss' or '::ss').

The same time information can be returned using DATEPART or DATENAME.

This function can also be invoked from ObjectScript using the **HOUR()** method call:

```
$SYSTEM.SQL.Functions.HOUR(time-expression)
```

Arguments

time-expression

An expression that is the name of a column, the result of another scalar function, or a string or numeric literal. It must resolve either to a datetime string or a time integer, where the underlying data type can be represented as %Time, %TimeStamp, or %PosixTime.

Examples

The following examples both return the number 18 because the *time-expression* value is 18:45:38:

SQL

```
SELECT {fn HOUR('2017-02-16 18:45:38')} AS ODBCHour
```

SQL

```
SELECT {fn HOUR(67538)} AS HorologHour
```

The following example also returns 18. The seconds (or minutes and seconds) portion of the time value can be omitted.

SQL

```
SELECT {fn HOUR('2017-02-16 18:45')} AS Hour_Given
```

The following example returns 0 hours, because the time portion of the datetime string has been omitted:

SQL

```
SELECT {fn HOUR('2017-02-16')} AS Hour_Given
```

The following examples all return the hours portion of the current time:

SQL

```
SELECT {fn HOUR(CURRENT_TIME)} AS H_CurrentT,
       {fn HOUR({fn CURTIME()})} AS H_CurT,
       {fn HOUR({fn NOW()})} AS H_Now,
       {fn HOUR($HOROLOG)} AS H_Horolog,
       {fn HOUR($ZTIMESTAMP)} AS H_ZTS
```

Note that **$ZTIMESTAMP** returns Coordinated Universal Time (UTC). The other *time-expression* values return the local time.

The following example shows that leading zeros are suppressed. The first HOUR function returns a length 2, the others return a length of 1. An omitted time is considered to be 0 hours, which has a length of 1:

SQL

```
SELECT LENGTH({fn HOUR('2018-02-15 11:45')}),
       LENGTH({fn HOUR('2018-02-15 03:45')}),
       LENGTH({fn HOUR('2018-02-15 3:45')}),
       LENGTH({fn HOUR('2018-02-15')})
```

The following example shows that the **HOUR** function recognizes the TimeSeparator character specified for the locale:

SQL

```
SELECT {fn HOUR('2018-02-16 18.45.38')}
```

See Also

- SQL concepts: Data Type, Date and Time Constructs

- SQL functions: MINUTE, SECOND, CURRENT_TIME, CURTIME, NOW, DATEPART, DATENAME

- ObjectScript function: $ZTIME

- ObjectScript special variables: $HOROLOG, $ZTIMESTAMP

IFNULL (SQL)

A function that tests for NULL and returns the appropriate expression.

Synopsis

```
IFNULL(expression-1,expression-2 [,expression-3])
```

```
{fn IFNULL(expression-1,expression-2)}
```

Description

InterSystems IRIS supports **IFNULL** as both an SQL general function and an ODBC scalar function. Note that while these two perform very similar operations, they are functionally different. The SQL general function supports three arguments. The ODBC scalar function supports two arguments. The two-argument forms of the SQL general function and the ODBC scalar function are not the same; they return different values when *expression-1* is not null.

The SQL general function evaluates whether *expression-1* is NULL. It never returns *expression-1*:

- If *expression-1* is NULL, *expression-2* is returned.

- If *expression-1* is not NULL, *expression-3* is returned.

- If *expression-1* is not NULL, and there is no *expression-3*, NULL is returned.

The ODBC scalar function evaluates whether *expression-1* is NULL. It either returns *expression-1* or *expression-2*:

- If *expression-1* is NULL, *expression-2* is returned.

- If *expression-1* is not NULL, *expression-1* is returned.

Refer to NULL for further details on NULL handling.

Data Type of Returned Value

- IFNULL(*expression-1*,*expression-2*): returns the data type of *expression-2*. If *expression-2* is a numeric literal, a string literal, or NULL returns data type VARCHAR.

- IFNULL(*expression-1*,*expression-2*,*expression-3*): if *expression-2* and *expression-3* have different data types, returns the data type with the higher (more inclusive) data type precedence. If *expression-2* or *expression-3* is a numeric literal or a string literal, returns data type VARCHAR. If *expression-2* or *expression-3* is NULL, returns the data type of the non-NULL argument.

  If *expression-2* and *expression-3* have different length, precision, or scale, **IFNULL** returns the greater length, precision, or scale of the two expressions.

- {fn IFNULL(*expression-1*,*expression-2*)}: returns the data type of *expression-1*. If *expression-1* is a numeric literal, a string literal, or NULL, returns data type VARCHAR.

DATE and TIME Display Conversion

Some *expression-1* data types, such as DATE and TIME data types, require conversion from Logical mode (mode 0) to ODBC mode (mode 1) or Display mode (mode 2). If the *expression-2* or *expression-3* value is not the same data type, this value cannot be converted in ODBC mode or Display mode, and an SQLCODE error is generated: -146 for DATE data type; -147 for TIME data type. For example, `IFNULL(DOB,'nodate',DOB)` cannot be executed in ODBC mode or Display mode; it issue an SQLCODE -146 error with the %msg `Error: 'nodate' is an invalid ODBC/JDBC Date value` or `Error: 'nodate' is an invalid DISPLAY Date value`. To execute this statement in ODBC mode or Display mode, you must CAST the value as the appropriate data type: `IFNULL(DOB,CAST('nodate' as DATE),DOB)`. This results in a date 0, which displays as 1840-12-31.

%List Display Conversion

A %List field is a string data type field with encoding. If *expression-1* is a %List field, the appropriate *expression-2* or *expression-3* value depends on the Select Mode:

- In Logical mode (mode 0) or Display mode (mode 2), a %List value is returned as string data type with the format `$lb("element1","element2",...)`. Therefore, an *expression-2* or *expression-3* value must be specified as a %List, as shown in the following example:

  SQL

  ```
  SELECT TOP 20 Name,
  IFNULL(FavoriteColors,$LISTBUILD('No Preference'),FavoriteColors) AS ColorChoice
  FROM Sample.Person
  ```

- In ODBC mode (mode 1), a %List value is returned as a string of comma-separated elements: `element1,element2,...`. Therefore, an *expression-2* or *expression-3* value can be specified as a string as shown in the following example:

  ObjectScript

  ```
  SELECT TOP 20 Name,
  IFNULL(FavoriteColors'No Preference',FavoriteColors) AS ColorChoice
  FROM Sample.Person
  ```

Arguments

expression-1

The expression to be evaluated to determine if it is NULL or not.

expression-2

An expression that is returned if *expression-1* is NULL.

expression-3

An optional expression that is returned if *expression-1* is not NULL. If *expression-3* is not specified, a NULL value is returned when *expression-1* is not NULL.

The returned data type is described below.

NULL Handling Functions Compared

The following table shows the various SQL comparison functions. Each function returns one value if the logical comparison tests True (A same as B) and another value if the logical comparison tests False (A not same as B). These functions allow you to perform NULL logical comparisons. You cannot specify NULL in an actual equality (or non-equality) condition comparison.

SQL Function	Comparison Test	Return Value
IFNULL(ex1,ex2) [two-argument form]	ex1 = NULL	True returns ex2 False returns NULL
IFNULL(ex1,ex2,ex3) [three-argument form]	ex1 = NULL	True returns ex2 False returns ex3
{fn IFNULL(ex1,ex2)}	ex1 = NULL	True returns ex2 False returns ex1
ISNULL(ex1,ex2)	ex1 = NULL	True returns ex2 False returns ex1
NVL(ex1,ex2)	ex1 = NULL	True returns ex2 False returns ex1
NULLIF(ex1,ex2)	ex1 = ex2	True returns NULL False returns ex1
COALESCE(ex1,ex2,...)	ex = NULL for each argument	True tests next ex argument. If all ex arguments are True (NULL), returns NULL. False returns ex

Examples

In the following example, the general function and the ODBC scalar function both returns the second expression (99) because the first expression is NULL:

SQL

```
SELECT IFNULL(NULL,99) AS NullGen,{fn IFNULL(NULL,99)} AS NullODBC
```

In the following example, the general function and the ODBC scalar function examples return different values. The general function returns <null> because the first expression is not NULL. The ODBC example returns the first expression (33) because the first expression is not NULL:

SQL

```
SELECT IFNULL(33,99) AS NullGen,{fn IFNULL(33,99)} AS NullODBC
```

The following example returns the string 'No Preference' if FavoriteColors is NULL; otherwise, it returns NULL:

SQL

```
SELECT Name,
IFNULL(FavoriteColors,'No Preference') AS ColorChoice
FROM Sample.Person
```

The following example returns the string 'No Preference' if FavoriteColors is NULL; otherwise, it returns the value of FavoriteColors:

SQL

```
SELECT Name,
IFNULL(FavoriteColors,'No Preference',FavoriteColors) AS ColorChoice
FROM Sample.Person
```

The following example returns the string 'No Preference' if FavoriteColors is NULL; otherwise, it returns the string 'Preference':

SQL

```
SELECT Name,
IFNULL(FavoriteColors,'No Preference','Preference') AS ColorPref
FROM Sample.Person
```

The following ODBC syntax examples return the string 'No Preference' if FavoriteColors is NULL, otherwise they return the FavoriteColors data value:

SQL

```
SELECT Name,
      {fn IFNULL(FavoriteColors,$LISTBUILD('No Preference'))} AS ColorPref
FROM Sample.Person
```

SQL

```
SELECT Name,
{fn IFNULL(FavoriteColors,'No Preference')} AS ColorChoice
FROM Sample.Person
```

The following example uses **IFNULL** in the **WHERE** clause. It selects people under the age of 21 who do not have favorite color preferences. If FavoriteColors is NULL, **IFNULL** returns the Age field value, which is used for the condition test:

SQL

```
SELECT Name,FavoriteColors,Age
FROM Sample.Person
WHERE 21 > IFNULL(FavoriteColors,Age)
ORDER BY Age
```

Refer to the NULL predicate (**IS NULL**, **IS NOT NULL**) for similar functionality.

See Also

- CASE command

- COALESCE function

- ISNULL function

- NULLIF function

- NVL function

- NULL predicate

INSTR (SQL)

A string function that returns the position of a substring within a string, with an optional search start point and occurrence count.

Synopsis

```
INSTR(string,substring[,start[,occurrence]])
```

Description

INSTR searches *string* for *substring*, and returns the position of the first character of *substring*. The position is returned as an integer, counting from the beginning of *string*. If *substring* is not found, 0 (zero) is returned. **INSTR** returns NULL if passed a NULL value for either argument.

INSTR supports specifying *start* as the starting point for substring search. **INSTR** also supports specifying the substring *occurrence* from that starting point.

INSTR is case-sensitive. Use one of the case-conversion functions to locate both uppercase and lowercase instances of a letter or character string.

This function can also be invoked from ObjectScript using the **INSTR()** method call:

ObjectScript

```
WRITE $SYSTEM.SQL.Functions.INSTR("The broken brown briefcase","br",6,2)
```

INSTR, CHARINDEX, POSITION, and $FIND

INSTR, **CHARINDEX**, **POSITION**, and **$FIND** all search a string for a specified substring and return an integer position corresponding to the first match. **CHARINDEX**, **POSITION**, and **INSTR** return the integer position of the first character of the matching substring. **$FIND** returns the integer position of the first character after the end of the matching substring. **CHARINDEX**, **$FIND**, and **INSTR** support specifying a starting point for substring search. **INSTR** also supports specifying the substring occurrence from that starting point.

The following example demonstrates these four functions, specifying all optional arguments. Note that the positions of *string* and *substring* differ in these functions:

SQL

```
SELECT POSITION('br' IN 'The broken brown briefcase') AS Position,
       CHARINDEX('br','The broken brown briefcase',6) AS Charindex,
       $FIND('The broken brown briefcase','br',6) AS Find,
       INSTR('The broken brown briefcase','br',6,2) AS Inst
```

For a list of functions that search for a substring, refer to String Manipulation.

Arguments

string

The string expression within which to search for *substring*. It can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type can be represented as any character type (such as CHAR or VAR-CHAR2).

substring

A substring that is believed to occur within *string*.

---

start

An optional argument specifying the starting point for substring search, specified as a positive integer. A character count from the beginning of *string*, counting from 1. To search from the beginning of *string*, omit this argument or specify a *start* of 1. A *start* value of 0, the empty string, NULL, or a nonnumeric value cause **INSTR** to return 0. Specifying *start* as a negative number causes **INSTR** to return <null>.

occurrence

An optional non-zero integer that specifies which occurrence of *substring* to return when searching from the *start* position. The default is to return the position of the first occurrence.

INSTR returns the INTEGER data type.

Examples

The following example returns 11, because "b" is the 11th character in the string:

SQL

```
SELECT INSTR('The quick brown fox','b',1) AS PosInt
```

The following example returns the length of the last name (surname) for each name in the Sample.Person table. It locates the comma used to separate the last name from the rest of the name field, then subtracts 1 from that position:

SQL

```
SELECT Name,
INSTR(Name,',',1)-1 AS LNameLen
FROM Sample.Person
```

The following example returns the position of the first instance of the letter "B" in each name in the Sample.Person table. Because **INSTR** is case-sensitive, the **%SQLUPPER** function is used to convert all name values to uppercase before performing the search. Because **%SQLUPPER** adds a blank space at the beginning of a string, this example subtracts 1 to get the actual letter position. Searches that do not locate the specified string return zero (0); in this example, because of the subtraction of 1, the value displayed for these searches is –1:

SQL

```
SELECT Name,
INSTR(%SQLUPPER(Name),'B',1)-1 AS BPos
FROM Sample.Person
```

See Also

* CHARINDEX function

* $FIND function

* POSITION function

* String Manipulation

%INTERNAL (SQL)

A format-transformation function that returns an expression in LOGICAL format.

Synopsis

```
%INTERNAL(expression)

%INTERNAL expression
```

Description

%INTERNAL converts *expression* to LOGICAL format, regardless of the current select mode (display mode). The LOGICAL format is the in-memory format of data (the format upon which operations are performed). **%INTERNAL** is commonly used on a **SELECT** list *select-item*.

%INTERNAL can be used in a **WHERE** clause, but this use is strongly discouraged because using **%INTERNAL** prevents the use of indexes on the specified field, and **%INTERNAL** forces all comparisons to be case-sensitive, even if the field has default collation.

Applying **%INTERNAL** changes the column header name to a value such as "Expression_1"; it is therefore usually desirable to specify a column name alias, as shown in the examples below.

%INTERNAL converts a value of data type %Date to an INTEGER data type value. **%INTERNAL** converts a value of data type %Time to a NUMERIC (15,9) data type value. This conversion is provided because an ODBC or JDBC client does not recognize InterSystems IRIS logical %Date and %Time values.

Whether **%INTERNAL** converts a date depends on the data type returned by the date field or function. **%INTERNAL** converts CURDATE, CURRENT_DATE, CURTIME, and CURRENT_TIME values. It does not convert CURRENT_TIMESTAMP, GETDATE, GETUTCDATE, NOW, and **$HOROLOG** values.

A stream field cannot be specified as an argument to ObjectScript unary functions, including all format-transformation functions, with the exception of **%INTERNAL**. The **%INTERNAL** function permits a stream field as an *expression* value, but performs no operation on that stream field.

%INTERNAL is an InterSystems SQL extension.

To convert an *expression* to DISPLAY format, regardless of the current select mode, use the %EXTERNAL function. To convert an *expression* to ODBC format, regardless of the current select mode, use the %ODBCOUT function.

For further details on display format options, refer to Data Display Options.

Arguments

expression

The expression to be converted. A field name, an expression containing a field name, or a function that returns a value in a convertible data type, such as DATE or %List.

Examples

The following example returns Date of Birth (DOB) data values in the current select mode format, and the same data using the **%INTERNAL** function:

SQL

```
SELECT TOP 5 DOB,%INTERNAL(DOB) AS IntDOB
FROM Sample.Person
```

The following examples show the two syntax forms for this function; they are otherwise identical. They specify the **%EXTERNAL** (DISPLAY format), **%INTERNAL** (LOGICAL format), and **%ODBCOUT** (ODBC format) of a %List field:

SQL

```
SELECT TOP 10 %EXTERNAL(FavoriteColors) AS ExtColors,
              %INTERNAL(FavoriteColors) AS IntColors,
              %ODBCOUT(FavoriteColors) AS ODBCColors
FROM Sample.Person
```

SQL

```
SELECT TOP 10 %EXTERNAL FavoriteColors AS ExtColors,
              %INTERNAL FavoriteColors AS IntColors,
              %ODBCOUT FavoriteColors AS ODBCColors
FROM Sample.Person
```

See Also

- %EXTERNAL, %ODBCIN, %ODBCOUT

- SQL concepts: Data Types, Date and Time Constructs

ISNULL (SQL)

A function that tests for NULL and returns the appropriate expression.

Synopsis

```
ISNULL(check-expression,replace-expression)
```

Arguments

Argument	Description
check-expression	The expression to be evaluated.
replace-expression	An expression that is returned if *check-expression* is NULL.

ISNULL returns the same data type as *check-expression*.

Description

ISNULL evaluates *check-expression* and returns one of two values:

• If *check-expression* is NULL, *replace-expression* is returned.

• If *check-expression* is not NULL, *check-expression* is returned.

The data type of *replace-expression* should be compatible with the data type of *check-expression*.

Note that the **ISNULL** function is the same as the **NVL** function, which is provided for Oracle compatibility.

Refer to NULL for further details on NULL handling.

DATE and TIME Display Conversion

Some *check-expression* data types require conversion from Logical mode to ODBC mode or Display mode, such as the DATE and TIME data types. If the *replace-expression* value is not the same data type, this value cannot be converted in ODBC mode or Display mode, and an SQLCODE error is generated: -146 for DATE data type; -147 for TIME data type. For example, `ISNULL(DOB,'nodate')` cannot be executed in ODBC mode or Display mode; it issues an SQLCODE -146 error with the %msg `Error: 'nodate' is an invalid ODBC/JDBC Date value` or `Error: 'nodate' is an invalid DISPLAY Date value`. To execute this statement in ODBC mode or Display mode, you must CAST the value as the appropriate data type: `ISNULL(DOB,CAST('nodate' as DATE))`. This results in a date 0, which displays as 1840-12-31.

NULL Handling Functions Compared

The following table shows the various SQL comparison functions. Each function returns one value if the logical comparison tests True (A same as B) and another value if the logical comparison tests False (A not same as B). These functions allow you to perform NULL logical comparisons. You cannot specify NULL in an actual equality (or non-equality) condition comparison.

SQL Function	Comparison Test	Return Value
ISNULL(ex1,ex2)	ex1 = NULL	True returns ex2 False returns ex1
IFNULL(ex1,ex2) [two-argument form]	ex1 = NULL	True returns ex2 False returns NULL
IFNULL(ex1,ex2,ex3) [three-argument form]	ex1 = NULL	True returns ex2 False returns ex3
{fn IFNULL(ex1,ex2)}	ex1 = NULL	True returns ex2 False returns ex1
NVL(ex1,ex2)	ex1 = NULL	True returns ex2 False returns ex1
NULLIF(ex1,ex2)	ex1 = ex2	True returns NULL False returns ex1
COALESCE(ex1,ex2,...)	ex = NULL for each argument	True tests next ex argument. If all ex arguments are True (NULL), returns NULL. False returns ex

Examples

In the following example, the first **ISNULL** returns the second expression (99) because the first expression is NULL. The second **ISNULL** returns the first expression (33) because the first expression is not NULL:

SQL

```
SELECT ISNULL(NULL,99) AS IsNullT,ISNULL(33,99) AS IsNullF
```

The following example returns the string 'No Preference' if FavoriteColors is NULL; otherwise, it returns the value of FavoriteColors:

SQL

```
SELECT Name,
ISNULL(FavoriteColors,'No Preference') AS ColorChoice
FROM Sample.Person
```

Compare the behavior of **ISNULL** with **IFNULL**:

SQL

```
SELECT Name,
IFNULL(FavoriteColors,'No Preference') AS ColorChoice
FROM Sample.Person
```

See Also

- CASE command
- COALESCE function
- IFNULL function
- NULLIF function
- NVL function

ISNUMERIC (SQL)

A numeric function that tests for a valid number.

Synopsis

```
ISNUMERIC(check-expression)
```

Description

ISNUMERIC evaluates *check-expression* and returns one of the following values:

- Returns 1 if *check-expression* is a valid number. A valid number can either be a numeric expression or a string that represents a valid number.

  - A numeric expression is first converted to canonical form, resolving multiple leading signs; therefore, a numeric expression such as +-+++34 is a valid number.

  - A numeric string is not converted before evaluation. A numeric string must have at most one leading sign to evaluate as a valid number. A numeric string with a trailing decimal point evaluates as a valid number.

- Returns 0 if *check-expression* is not a valid number. Any string that contains a non-numeric character is not a valid number. A numeric string with more than one leading sign, such as '+-+++34', is not evaluated as a valid number. An InterSystems IRIS encoded list always returns 0, even if its element(s) are valid numbers. An empty string ISNUMERIC('') returns 0.

- Returns NULL if *check-expression* is NULL. ISNUMERIC(NULL) returns null.

ISNUMERIC generates an SQLCODE -7, exponent out of range error if a scientific notation exponent is greater than 308 (308 – (number of integers - 1)). For example, ISNUMERIC(1E309) and ISNUMERIC(111E307) both generate this error code. If an exponent numeric string less than or equal to '1E145' returns 1; an exponent numeric string greater than '1E145' returns 0.

The **ISNUMERIC** function is very similar to the ObjectScript **$ISVALIDNUM** function. However, these two functions return different values when the input value is NULL.

Arguments

check-expression

The expression to be evaluated.

Examples

In the following example, all of the **ISNUMERIC** functions return 1:

SQL

```
SELECT ISNUMERIC(99) AS MyInt,
       ISNUMERIC('-99') AS MyNegInt,
       ISNUMERIC('-0.99') AS MyNegFrac,
       ISNUMERIC('-0.00') AS MyNegZero,
       ISNUMERIC('-0.09'+7) AS MyAdd,
       ISNUMERIC('5E2') AS MyExponent
```

The following example returns NULL if FavoriteColors is NULL; otherwise, it returns 0, because FavoriteColors is not a numeric field:

SQL

```
SELECT Name,
ISNUMERIC(FavoriteColors) AS ColorPref
FROM Sample.Person
```

See Also

- IFNULL function

- ISNULL function

- NULLIF function

- ObjectScript function: $ISVALIDNUM

JSON_ARRAY (SQL)

A conversion function that returns data as a JSON array.

Synopsis

```
JSON_ARRAY(expression [,expression][,...] [NULL ON NULL | ABSENT ON NULL])
```

Description

JSON_ARRAY takes an expression or (more commonly) a comma-separated list of expressions and returns a JSON array containing those values. **JSON_ARRAY** can be combined in a **SELECT** statement with other types of select-items. **JSON_ARRAY** can be specified in other locations where an SQL function can be used, such as in a **WHERE** clause.

The returned JSON array has the following format:

```
[ element1 , element2 , element3 ]
```

JSON_ARRAY returns each array element value as either a string (enclosed in double quotes), or a number. Numbers are returned in canonical format. A numeric string is returned as a literal, enclosed in double quotes. All other data types (for example, Date or $List) are returned as a string.

JSON_ARRAY does not support asterisk (*) syntax as a way to specify all fields in a table. It does support the **COUNT(*)** aggregate function.

The returned JSON array column is labeled as an Expression (by default); you can specify a column alias for a **JSON_ARRAY**.

Select Mode and Collation

The current %SelectMode property determines the format of the returned JSON array values. By changing the Select Mode, all Date and %List elements are included in the JSON array as strings with that Select Mode format.

You can override the current Select Mode by applying a format-transformation function (%EXTERNAL, %INTERNAL, %ODBCIN, %ODBCOUT) to individual field names within **JSON_ARRAY**. Applying a format-transformation function to a **JSON_ARRAY** has no effect, because the elements of a JSON array are strings.

You can apply a collation function to individual field names within **JSON_ARRAY** or to an entire **JSON_ARRAY**:

- A collation function applied to a **JSON_ARRAY** applies the collation after JSON array formatting. Therefore, `%SQLUPPER(JSON_ARRAY(f1,f2))` converts all the JSON array element values to uppercase. `%SQLUPPER(JSON_ARRAY(f1,f2))` inserts a space before the JSON array, not before the elements of the array; therefore it does not force numbers to be parsed as strings.

- A collation function applied to an element within a **JSON_ARRAY** applies that collation. Therefore `JSON_ARRAY('Abc',%SQLUPPER('Abc'))` returns `["Abc"," ABC"]` (note leading space); and `JSON_ARRAY(007,%SQLSTRING(007))` returns `[7," 7"]`. Because %SQLUPPER inserts a space before the value, it is generally preferable to specify a case transformation function such as LCASE or UCASE. You can apply collation to both an element and to the whole array: `%SQLUPPER(JSON_ARRAY('Abc',%SQLSTRING('Abc')))` returns `["ABC"," ABC"]`

ABSENT ON NULL

If you specify the optional ABSENT ON NULL keyword phrase, a column value which is NULL (or the NULL literal) is not included in the JSON array. No placeholder is included in the JSON array. This can result in JSON arrays with different

numbers of elements. For example, the following program returns JSON arrays where for some records the 3rd array element is Age, and for other records the 3rd element is FavoriteColors:

```
SELECT JSON_ARRAY(%ID,Name,FavoriteColors,Age ABSENT ON NULL) FROM Sample.Person
```

If you specify no keyword phrase, the default is NULL ON NULL: NULL is represented by the word null (not delimited by quotes) as a comma-separated array element. Thus all JSON arrays returned by a **JSON_ARRAY** function will have the same number of array elements.

Arguments

expression

An expression or a comma-separated list of expressions. These expressions can include column names, aggregate functions, arithmetic expressions, literals, and the literal NULL.

ABSENT ON NULL/NULL ON NULL

An optional keyword phrase specifying how to represent NULL values in the returned JSON array. NULL ON NULL (the default) represents NULL (absent) data with the word null (not quoted). ABSENT ON NULL omits NULL data from the JSON array; it does not leave a placeholder comma. This keyword phrase has no effect on empty string values.

Examples

The following example applies **JSON_ARRAY** to format a JSON array containing a comma-separated list of field values:

SQL

```
SELECT TOP 3 JSON_ARRAY(%ID,Name,Age,Home_State) FROM Sample.Person
```

The following example applies **JSON_ARRAY** to format a JSON array with a single element containing the Name field values:

SQL

```
SELECT TOP 3 JSON_ARRAY(Name) FROM Sample.Person
```

The following example applies **JSON_ARRAY** to format a JSON array containing literals and field values:

SQL

```
SELECT TOP 3 JSON_ARRAY('Employee from',%TABLENAME,Name,SSN) FROM Sample.Employee
```

The following example applies **JSON_ARRAY** to format a JSON array containing nulls and field values:

SQL

```
SELECT JSON_ARRAY(Name,FavoriteColors) FROM Sample.Person
WHERE Name %STARTSWITH 'S'
```

The following example applies **JSON_ARRAY** to format a JSON array containing field values from joined tables:

SQL

```
SELECT TOP 3 JSON_ARRAY(E.%TABLENAME,E.Name,C.%TABLENAME,C.Name)
FROM Sample.Employee AS E,Sample.Company AS C
```

The following Dynamic SQL example sets the ODBC %SelectMode, which determines how all fields, including JSON array values are represented. The query overrides this Select Mode for specific JSON array elements by applying the **%EXTERNAL** format-transformation function:

ObjectScript

```
  SET myquery = 3
    SET myquery(1) = "SELECT TOP 8 DOB,JSON_ARRAY(Name,DOB,FavoriteColors) AS ODBCMode, "
    SET myquery(2) = "JSON_ARRAY(Name,DOB,%EXTERNAL(DOB),%EXTERNAL(FavoriteColors)) AS ExternalTrans
"
    SET myquery(3) = "FROM Sample.Person"
  SET tStatement = ##class(%SQL.Statement).%New()
  SET tStatement.%SelectMode=1
   WRITE "SelectMode is ODBC",!
  SET qStatus = tStatement.%Prepare(.myquery)
   IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET rset = tStatement.%Execute()
    IF rset.%SQLCODE=0 { WRITE !,"Executed query",! }
    ELSE { SET badSQL=##class(%Exception.SQL).%New(,rset.%SQLCODE,,rset.%Message)}
  DO rset.%Display()
  WRITE !,"End of data"
```

The following example uses **JSON_ARRAY** in a **WHERE** clause to perform a Contains test on multiple columns without using OR syntax:

SQL

```
SELECT Name,Home_City,Home_State FROM Sample.Person
WHERE JSON_ARRAY(Name,Home_City,Home_State) [ 'X'
```

See Also

- SELECT statement

- WHERE clause

- JSON_OBJECT function

- IS JSON predicate condition

- Overview of Predicates

- Querying the Database

JSON_OBJECT (SQL)

A conversion function that returns data as a JSON object.

Synopsis

```
JSON_OBJECT(key:value [,key:value][,...] [NULL ON
NULL | ABSENT ON NULL])
```

Description

JSON_OBJECT takes a comma-separated list of *key:value* pairs (for example, `'mykey':colname`) and returns a JSON object containing those values. You can specify any single-quoted string as a key name; **JSON_OBJECT** does not enforce any naming conventions or uniqueness check for key names. You can specify for value a column name or other expression.

JSON_OBJECT can be combined in a **SELECT** statement with other types of select-items. **JSON_OBJECT** can be specified in other locations where an SQL function can be used, such as in a **WHERE** clause.

A returned JSON object has the following format:

```
{ "key1" : "value1" , "key2" : "value2" , "key3" : "value3"
}
```

JSON_OBJECT returns object values as either a string (enclosed in double quotes), or a number. Numbers are returned in canonical format. A numeric string is returned as a literal, enclosed in double quotes. All other data types (for example, Date or $List) are returned as a string, with the current %SelectMode determining the format of the returned value. **JSON_OBJECT** returns both key and value values in DISPLAY or ODBC mode if that is the select mode for the query.

JSON_OBJECT does not support asterisk (*) syntax as a way to specify all fields in a table.

The returned JSON object column is labeled as an Expression (by default); you can specify a column alias for a **JSON_OBJECT**.

Select Mode and Collation

The current %SelectMode property determines the format of the returned JSON object values. By changing the Select Mode, all Date and %List values are included in the JSON object as strings with that Select Mode format. You can override the current Select Mode by applying a format-transformation function (%EXTERNAL, %INTERNAL, %ODBCIN, %ODBCOUT) to individual field names within **JSON_OBJECT**. Applying a format-transformation function to a **JSON_OBJECT** has no effect, because the key:value pairs of a JSON object are strings.

The default collation determines the collation of the returned JSON object values. You can apply a collation function to a **JSON_OBJECT**, converting both keys and values. Generally, you should not apply a collation function to **JSON_OBJECT** because keys are case-sensitive. InterSystems IRIS applies the collation after JSON object formatting. Therefore, `%SQLUPPER(JSON_OBJECT('k1':f1,'k2':f2))` converts all the JSON object key and value strings to uppercase. %SQLUPPER inserts a space before the JSON object, not before the values within the object.

Within **JSON_OBJECT**, you can apply a collation function to the value portion of a key:value pair. Because %SQLUPPER inserts a space before the value, it is generally preferable to specify a case transformation function such as LCASE or UCASE.

ABSENT ON NULL

If you specify the optional ABSENT ON NULL keyword phrase, a column value which is NULL (or the NULL literal) is not included in the JSON object. No placeholder is included in the JSON object. This can result in JSON objects with dif-

ferent numbers of key:value pairs. For example, the following program returns JSON objects where for some records the 3rd key:value pair is Age, and for other records the 3rd key:value pair is FavoriteColors:

```
SELECT JSON_OBJECT('id':%ID,'name':Name,'colors':FavoriteColors,'years':Age ABSENT ON NULL) FROM
Sample.Person
```

If you specify no keyword phrase, the default is NULL ON NULL: NULL is represented by the word null (not delimited by quotes) as the value of the key:value pair. Thus all JSON objects returned by a **JSON_OBJECT** function will have the same number of key:value pairs.

Arguments

key:value

A key:value pair or a comma-separated list of key:value pairs. A *key* is a user-specified literal string delimited with single quotes. A *value* can be a column name, an aggregate function, an arithmetic expression, a numeric or string literal, or the literal NULL.

ABSENT ON NULL/NULL ON NULL

An optional keyword phrase specifying how to represent NULL values in the returned JSON object. NULL ON NULL (the default) represents NULL (absent) data with the word null (not quoted). ABSENT ON NULL omits NULL data from the JSON object; it removes the key:value pair when value is NULL and does not leave a placeholder comma. This keyword phrase has no effect on empty string values.

Examples

This example applies **JSON_OBJECT** to format a JSON object containing field values:

SQL

```
SELECT TOP 3 JSON_OBJECT('id':%ID,'name':Name,'birth':DOB) FROM Sample.Person
```

This example applies **JSON_OBJECT** to format a JSON object containing literals and field values:

SQL

```
SELECT TOP 3 JSON_OBJECT('lit':'Employee from','t':%TABLENAME,
    'name':Name,'num':SSN) FROM Sample.Employee
```

This example applies **JSON_OBJECT** to format a JSON object containing nulls and field values:

SQL

```
SELECT JSON_OBJECT('name':Name,'colors':FavoriteColors) FROM Sample.Person
WHERE Name %STARTSWITH 'S'
```

This example applies **JSON_OBJECT** to format a JSON object containing field values from joined tables:

SQL

```
SELECT TOP 3 JSON_OBJECT('e.t':E.%TABLENAME,'e.name':E.Name,'c.t':C.%TABLENAME,
'c.name':C.Name) FROM Sample.Employee AS E,Sample.Company AS C
```

The following example uses **JSON_OBJECT** in a **WHERE** clause to perform a Contains test on multiple columns without using OR syntax:

SQL

```
SELECT Name,Home_City,Home_State FROM Sample.Person
WHERE JSON_OBJECT('name':Name,'city':Home_City,'state':Home_State) [ 'X'
```

The following Dynamic SQL example sets the ODBC %SelectMode, which determines how all fields, including JSON object values are represented. The query overrides this Select Mode for specific **JSON_OBJECT** values by applying the **%EXTERNAL** format-transformation function:

ObjectScript

```
  SET myquery = 3
    SET myquery(1) = "SELECT TOP 8 JSON_OBJECT('ODBCBday':DOB,'DispBday':%EXTERNAL(DOB)),"
    SET myquery(2) = "JSON_OBJECT('ODBCcolors':FavoriteColors,'DispColors':%EXTERNAL(FavoriteColors))
"
    SET myquery(3) = "FROM Sample.Person"
  SET tStatement = ##class(%SQL.Statement).%New()
  SET tStatement.%SelectMode=1
    WRITE "SelectMode is ODBC",!
  SET qStatus = tStatement.%Prepare(.myquery)
   IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET rset = tStatement.%Execute()
     IF rset.%SQLCODE=0 { WRITE !,"Executed query",! }
     ELSE { SET badSQL=##class(%Exception.SQL).%New(,rset.%SQLCODE,,rset.%Message)}
  DO rset.%Display()
  WRITE !,"End of data"
```

See Also

- SELECT statement

- WHERE clause

- JSON_ARRAY function

- IS JSON predicate condition

- Overview of Predicates

- Querying the Database

$JUSTIFY (SQL)

A function that right-aligns a value within a specified width, optionally rounding to a specified number of fractional digits.

Synopsis

```
$JUSTIFY(expression,width[,decimal])
```

Description

$JUSTIFY returns the value specified by *expression* right-aligned within the specified *width*. You can include the *decimal* argument to decimal-align numbers within *width*.

- `$JUSTIFY(expression,width)`: the 2-argument syntax right-justifies *expression* within *width*. It does not perform any conversion of *expression*. The *expression* can be a numeric or a nonnumeric string.

- `$JUSTIFY(expression,width,decimal)`: the 3-argument syntax converts *expression* to a canonical number, rounds or zero pads fractional digits to *decimal*, then right-justifies the resulting numeric value within *width*. If *expression* is a nonnumeric string or NULL, InterSystems IRIS converts it to 0, pads it, then right-justifies it.

$JUSTIFY recognizes the DecimalSeparator character for the current locale. It adds or deletes a DecimalSeparator character as needed. The DecimalSeparator character depends upon the locale; commonly it is either a period (.) for American-format locales, or a comma (,) for European-format locales. To determine the DecimalSeparator character for your locale, invoke the following method:

ObjectScript

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("DecimalSeparator")
```

SQLCODE -380 is issued if you specify too few arguments. SQLCODE -381 is issued if you specify too many arguments.

$JUSTIFY, ROUND, and TRUNCATE

When rounding to a fixed number of fractional digits is important — for example, when representing monetary amounts — use **$JUSTIFY**, which returns the specified number of trailing zeros following the rounding operation. When *decimal* is larger than the number of fractional digits in *expression*, **$JUSTIFY** zero-pads. **$JUSTIFY** also right-aligns the numbers, so that the DecimalSeparator characters align in a column of numbers.

ROUND also rounds to a specified number of fractional digits, but its return value is always normalized, removing trailing zeros. For example, `ROUND(10.004,2)` returns 10, not 10.00. Unlike **$JUSTIFY**, **ROUND** allows you to specify either rounding (the default), or truncation.

TRUNCATE truncates to a specified number of fractional digits. Unlike **ROUND**, if the truncation results in trailing zeros, these trailing zeros are preserved. However, unlike **$JUSTIFY**, **TRUNCATE** does not zero-pad.

ROUND and **TRUNCATE** allow you to round (or truncate) to the left of the decimal separator. For example, `ROUND(128.5,-1)` returns 130.

$JUSTIFY and LPAD

The two-argument form of **LPAD** and the two-argument form of **$JUSTIFY** both right-align a string by padding it with leading spaces. These two-argument forms differ in how they handle an output *width* that is shorter than the length of the input *expression*: **LPAD** truncates the input string to fit the specified output length. **$JUSTIFY** expands the output length to fit the input string. This is shown in the following example:

SQL

```
SELECT '>'||LPAD(12345,10)||'<' AS lpadplus,
       '>'||$JUSTIFY(12345,10)||'<' AS justifyplus,
       '>'||LPAD(12345,3)||'<' AS lpadminus,
       '>'||$JUSTIFY(12345,3)||'<' AS justifyminus
```

The three-argument form of **LPAD** allows you to left pad with characters other than spaces.

Arguments

expression

The value to be right-justified, and optionally expressed as a numeric with a specified number of fractional digits.

- If string justification is desired, do not specify *decimal*. The *expression* can contain any characters. **$JUSTIFY** right-justifies *expression*, as described in *width*.

- If numeric justification is desired, specify *decimal*. If *decimal* is specified, InterSystems IRIS supplies *expression* to **$JUSTIFY** as a canonical number. It resolves leading plus and minus signs and removes leading and trailing zeros. It truncates *expression* at the first nonnumeric character. If *expression* begins with a nonnumeric character (such as a currency symbol), InterSystems IRIS converts the *expression* value to 0. Canonical conversion does not recognize NumericGroupSeparator characters, currency symbols, multiple DecimalSeparator characters, or trailing plus or minus signs. For further details on how InterSystems IRIS converts a numeric to a canonical number, and InterSystems IRIS handling of a numeric string containing nonnumeric characters, refer to Numbers.

  After **$JUSTIFY** receives *expression* as a canonical number, **$JUSTIFY** performs its operation and either rounds or zero-pads this canonical number to *decimal* number of fractional digits, then right-justifies the result, as described in *width*.

width

The *width* in which to right-justify the converted *expression*. If *width* is greater than the length of *expression* (after numeric and fractional digit conversion), InterSystems IRIS right-justifies to width, left-padding as needed with blank spaces. If *width* is less than the length of *expression* (after numeric and fractional digit conversion), InterSystems IRIS sets *width* to the length of the *expression* value.

Specify *width* as a positive integer. A *width* value of 0, the empty string (''), NULL, or a nonnumeric string is treated as a *width* of 0, which means that InterSystems IRIS sets *width* to the length of the *expression* value.

decimal

The number of fractional digits. If *expression* contains more fractional digits, **$JUSTIFY** rounds the fractional portion to this number of fractional digits. If *expression* contains fewer fractional digits, **$JUSTIFY** pads the fractional portion with zeros to this number of fractional digits, adding a Decimal Separator character, if needed. If *decimal*=0, **$JUSTIFY** rounds *expression* to an integer value and deletes the Decimal Separator character.

If the *expression* value is less than 1, **$JUSTIFY** inserts a leading zero before the DecimalSeparator character.

The $DOUBLE values INF, -INF, and NAN are returned unchanged by **$JUSTIFY**, regardless of the *decimal* value.

Examples

The following example performs right-justification on strings. No numeric conversion is performed:

SQL

```
SELECT TOP 20 Age,$JUSTIFY(Name,18),DOB FROM Sample.Person
```

The following example performs numeric right-justification with a specified number of fractional digits:

SQL

```
SELECT TOP 20 $JUSTIFY(Salary,10,2) AS FullSalary,
$JUSTIFY(Salary/7,10,2) AS SeventhSalary FROM Sample.Employee
```

The following example performs numeric right-justification with a specified number of fractional digits, and string right-justification of the same numeric value:

ObjectScript

```
"SELECT $JUSTIFY({fn ACOS(-1)},8,3) AS ArcCos3,
$JUSTIFY({fn ACOS(-1)},8) AS ArcCosAll
```

The following Dynamic SQL example performs numeric right-justification with the **$DOUBLE** values INF and NAN:

ObjectScript

```
DO ##class(%SYSTEM.Process).IEEEError(0)
SET x=$DOUBLE(1.2e500)
SET y=x-x
SET myquery = 2
SET myquery(1) = "SELECT $JUSTIFY(?,12,2) AS INFtest,"
SET myquery(2) = "$JUSTIFY(?,12,2) AS NANtest"
 SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(x,y)
DO rset.%Display()
```

See Also

- **LPAD** function

- **ROUND** function

- **TRUNCATE** function

LAST_DAY (SQL)

A date function that returns the date of the last day of the month for a date expression.

Synopsis

```
LAST_DAY(date-expression)
```

Description

LAST_DAY returns the date of the last day of the specified month as an integer in $HOROLOG format. Leap year differences are calculated, including century day adjustments: 2000 is a leap year, 1900 and 2100 are not leap years.

The *date-expression* can be an InterSystems IRIS date integer, a $HOROLOG or $ZTIMESTAMP value, an ODBC format date string, or a timestamp.

A *date-expression* timestamp can be either data type %Library.PosixTime (an encoded 64-bit signed integer), or data type %Library.TimeStamp (yyyy-mm-dd hh:mm:ss.fff).

The time portion of a %TimeStamp string is optional.

LAST_DAY returns 0 (in Display mode `12/31/1840`) when an invalid date is specified: the day or month as zero; the month greater than 12; or the day larger than the number of days in that month on that year. The year must be in the range 0001 through 9999.

This function can also be invoked from ObjectScript using the **LASTDAY()** method call:

ObjectScript

```
WRITE $SYSTEM.SQL.Functions.LASTDAY("2018-02-22"),!
WRITE $SYSTEM.SQL.Functions.LASTDAY(64701)
```

Arguments

date-expression

An expression that is the name of a column, the result of another scalar function, or a date or timestamp literal.

Examples

The following examples return the last day of the month as an InterSystems IRIS date integer. Whether this value is displayed as an integer or as a date string depends on the current SQL Display Mode setting.

The following two examples both return the number 59594 (which corresponds to '2004–02–29') because the last day of the month on the specified date is February 29 (2004 is a leap year):

SQL

```
SELECT LAST_DAY('2004-02-25')
```

SQL

```
SELECT LAST_DAY(59590)
```

The following examples all return the InterSystems IRIS date integer corresponding to the last day of the current month:

SQL

```
SELECT LAST_DAY({fn NOW()}) AS LD_Now,
       LAST_DAY(CURRENT_DATE) AS LD_CurrDate,
       LAST_DAY(CURRENT_TIMESTAMP) AS LD_CurrTstamp,
       LAST_DAY($ZTIMESTAMP) AS LD_ZTstamp,
       LAST_DAY($HOROLOG) AS LD_Horolog
```

See Also

- SQL functions: DATENAME, DATEPART, DAY, DAYOFYEAR, MONTH, YEAR, TO_DATE

- ObjectScript function: $ZDATE

- ObjectScript special variables: $HOROLOG, $ZTIMESTAMP

LAST_IDENTITY (SQL)

A scalar function that returns the identity of the last row inserted, updated, deleted, or fetched.

Synopsis

```
LAST_IDENTITY()
```

Description

The **LAST_IDENTITY** function returns the %ROWID local variable value. The %ROWID local variable is set to a value in Embedded SQL or ODBC. The %ROWID local variable is not set to a value by Dynamic SQL, the SQL Shell, or the Management Portal SQL interface. Dynamic SQL instead sets a *%ROWID* object property.

The **LAST_IDENTITY** function takes no arguments. Note that the argument parentheses are required.

LAST_IDENTITY returns the IDENTITY field value of the last row affected by the current process. If the table has no IDENTITY field, it returns the row ID (%ROWID) of the last row affected by the current process. The returned value is data type INTEGER.

- For an Embedded SQL INSERT, UPDATE, DELETE or TRUNCATE TABLE statement, **LAST_IDENTITY** returns the IDENTITY or %ROWID value of the last row modified.

- For an Embedded SQL cursor-based SELECT statement, **LAST_IDENTITY** returns the IDENTITY or %ROWID value of the last row retrieved. However, if the cursor-based **SELECT** statement includes a DISTINCT keyword or a GROUP BY clause, **LAST_IDENTITY** is not changed; it returns its prior value (if any).

- For an Embedded SQL single-row (non-cursor) **SELECT** statement, **LAST_IDENTITY** is not changed. The prior value (if any) is returned.

At process initiation, **LAST_IDENTITY** returns NULL. Following a **NEW %RowID**, **LAST_IDENTITY** returns NULL.

If no rows were affected by an operation, **LAST_IDENTITY** is not changed; **LAST_IDENTITY** returns its prior value (if any). Following a **NEW %RowID**, invoking **LAST_IDENTITY** returns NULL, but invoking %ROWID generates an <UNDEFINED> error.

For further details on IDENTITY fields, see CREATE TABLE. Also see %ROWID.

Examples

The following example uses two Embedded SQL programs to return **LAST_IDENTITY**. The first example creates a new table Sample.Students, The second example populates this table with data, then performs a cursor-based **SELECT** on the data, returning **LAST_IDENTITY** for each operation.

Please run the two Embedded SQL programs in the order shown. (It is necessary to use two embedded SQL programs here because embedded SQL cannot compile an **INSERT** statement unless the referenced table already exists.)

ObjectScript

```
  WRITE !,"Creating table"
&sql(CREATE TABLE Sample.Students (
  StudentName VARCHAR(30),
  StudentAge INTEGER,
  StudentID IDENTITY))
IF SQLCODE=0 {
  WRITE !,"Created table, SQLCODE=",SQLCODE }
ELSEIF SQLCODE=-201 {
  WRITE !,"Table already exists, SQLCODE=",SQLCODE }
```

ObjectScript

```
WRITE !,"Populating table"
NEW %ROWCOUNT,%ROWID
&sql(INSERT INTO Sample.Students (StudentName,StudentAge)
    SELECT Name,Age FROM Sample.Person WHERE Age <= '21')
IF SQLCODE=0 {
  WRITE !,%ROWCOUNT," records added, last RowID is ",%ROWID,! }
ELSE {
  WRITE !,"Insert failed, SQLCODE=",SQLCODE }
&sql(SELECT LAST_IDENTITY()
     INTO :insertID
     FROM Sample.Students)
  WRITE !,"INSERT Last Identity is: ",insertID,!!
 /* Cursor-based SELECT Query */
     &sql(DECLARE C1 CURSOR FOR
     SELECT StudentName INTO :name FROM  Sample.Students
     WHERE StudentAge = '17')
&sql(OPEN C1)
     QUIT:(SQLCODE'=0)
&sql(FETCH C1)
WHILE (SQLCODE = 0) {
   WRITE name," is seventeen",!
  &sql(FETCH C1) }
&sql(CLOSE C1)
 WRITE !,%ROWCOUNT," records queried, last RowID is ",%ROWID,!
&sql(SELECT LAST_IDENTITY()
     INTO :qId)
  WRITE !,"SELECT Last Identity is: ",qId,!
&sql(DROP TABLE Sample.Students)
```

See Also

- INSERT, UPDATE, DELETE, TRUNCATE TABLE

- DECLARE, OPEN, FETCH, CLOSE

- Embedded SQL

LCASE (SQL)

A case-transformation function that converts all uppercase letters in a string to lowercase letters.

Synopsis

```
LCASE(string-expression)
```

```
{fn LCASE(string-expression)}
```

Description

LCASE converts uppercase letters to lowercase for display purposes. It has no effects on non-alphabetic characters. It leaves unchanged punctuation and leading and trailing blank spaces.

LCASE does not force numerics to be interpreted as a string. InterSystems SQL converts numerics to canonical form, removing leading and trailing zeros. InterSystems SQL does not convert numeric strings to canonical form.

The **LOWER** function can also be used convert uppercase letters to lowercase.

LCASE does not affect collation. The **%SQLUPPER** function is the preferred way in SQL to convert a data value for not case-sensitive collation. Refer to %SQLUPPER for further information on case transformation for collation.

Arguments

string-expression

The string expression whose characters are to be converted to lowercase. The expression can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR).

Examples

The following example returns each person's name in lowercase letters:

SQL

```
SELECT TOP 10 Name,{fn LCASE(Name)} AS LowName
    FROM Sample.Person
```

LCASE also works on Unicode (non-ASCII) alphabetic characters, as shown in the following example, which converts Greek letters from uppercase to lowercase:

SQL

```
SELECT LCASE($CHAR(920,913,923,913,931,931,913))
```

See Also

- SQL functions: LOWER, UCASE
- ObjectScript function: $ZCONVERT

LEAST (SQL)

A function that returns the least value from a series of expressions.

Synopsis

```
LEAST(expression,expression[,...])
```

Description

LEAST returns the smallest (least) value from a comma-separated series of expressions. Expressions are evaluated in left-to-right order. If only one *expression* is provided, **LEAST** returns that value. If any *expression* is NULL, **LEAST** returns NULL.

If all of the *expression* values resolve to canonical numbers, they are compared in numeric order. If a quoted string contains a number in canonical format, it is compared in numeric order. However, if a quoted string contains a number not in canonical format (for example, '00', '0.4', or '+4'), it is compared as a string. String comparisons are performed character-by-character in collation order. Any string value is greater than any numeric value.

The empty string is greater than any numeric value, but less than any other string value.

If the returned value is a number, **LEAST** returns it in canonical format (leading and trailing zeros removed, etc.). If the returned value is a string, **LEAST** returns it unchanged, including any leading or trailing blanks.

LEAST returns the least value from a comma-separated series of expressions. GREATEST returns the greatest value from a comma-separated series of expressions. COALESCE returns the first non-NULL value from a comma-separated series of expressions.

Data Type of Returned Value

If the data types of the *expression* values are different, the data type returned is the type most compatible with all of the possible return values, the data type with the highest data type precedence. For example, if one *expression* is an integer and another *expression* is a fractional number, **LEAST** returns a value with data type NUMERIC. This is because NUMERIC is the data type with the highest precedence that is compatible with both. If, however, an *expression* is a literal number or string, **LEAST** returns data type VARCHAR.

Arguments

expression

An expression that resolves to a number or a string. The values of these expressions are compared to each other and the least value returned. An *expression* can be a field name, a literal, an arithmetic expression, a host variable, or an object reference. You can list up to 140 comma-separated expressions.

Examples

In the following example, each **LEAST** compares three canonical numbers:

SQL

```
SELECT LEAST(22,2.2,-21) AS HighNum,
       LEAST('2.2','22','-21') AS HighNumStr
```

In the following example, each **LEAST** compare three numeric strings. However, each **LEAST** contains one string that is non-canonical; these non-canonical values are compared as character strings. A character string is always greater than a number:

SQL

```
SELECT LEAST('22','+2.2','21'),
       LEAST('0.2','22','21')
```

In the following example, each **LEAST** compare three strings and returns the value with the lowest collation sequence:

SQL

```
SELECT LEAST('A','a',''),
       LEAST('a','aa','abc'),
       LEAST('#','0','7'),
       LEAST('##','00','77')
```

The following example compares two dates, treated as canonical numbers: the date of birth as a $HOROLOG integer, and the integer 58074 converted to a date. It returns the date of birth for each person born in the 20th century. Anyone born after December 31, 1999 is displayed with the default birth date of January 1, 2000:

SQL

```
SELECT Name,LEAST(DOB,TO_DATE(58074)) AS NewMillenium
FROM Sample.Person
```

See Also

- GREATEST function

- COALESCE function

- CONVERT function

- TO_NUMBER function

LEFT (SQL)

A scalar string function that returns a specified number of characters from the beginning (leftmost position) of a string expression.

Synopsis

```
{fn LEFT(string-expression,count)}
```

Description

LEFT returns the specified number of characters from the beginning of a string. **LEFT** does not pad strings; if you specify a larger number of characters than are in the string, **LEFT** returns the string. **LEFT** returns NULL if passed a NULL value for either argument.

LEFT can only be used as an ODBC scalar function (with the curly brace syntax).

Arguments

string-expression

A string expression, which can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR).

count

An integer that specifies the number of characters to return from the starting position of *string-expression*.

Examples

The following example returns the seven leftmost characters from each name in the Sample.Person table:

SQL

```
SELECT Name,{fn LEFT(Name,7)}AS ShortName
     FROM Sample.Person
```

The following example shows how **LEFT** handles a *count* that is longer than the string itself:

SQL

```
SELECT Name,{fn LEFT(Name,40)}
     FROM Sample.Person
```

No padding is performed.

See Also

LTRIM RIGHT RTRIM

LEN (SQL)

A string function that returns the number of characters in a string expression.

Synopsis

```
LEN(string-expression)
```

Description

LEN returns the number of characters in a string expression

The **LEN** function is an alias for the **LENGTH** function. **LEN** is provided for TSQL compatibility. Refer to LENGTH for further details.

Arguments

string-expression

A string expression, which can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR).

LEN returns the INTEGER data type.

See Also

* LENGTH

LENGTH (SQL)

A string function that returns the number of characters in a string expression.

Synopsis

```
LENGTH(string-expression)
```

```
{fn LENGTH(string-expression)}
```

Description

LENGTH returns an integer that denotes the number of characters, not the number of bytes, of the given string expression. The *string-expression* can be a string (from which trailing blanks are removed), or a number (which InterSystems IRIS converts to canonical form).

Note that **LENGTH** can be used as an ODBC scalar function (with the curly brace syntax) or as an SQL general function.

LENGTH and the other length functions (**$LENGTH**, **CHARACTER_LENGTH**, **CHAR_LENGTH**, and **DATALENGTH**) all perform the following operations:

- **LENGTH** returns the length of the Logical (internal data storage) value of a field, not the display value, regardless of the SelectMode setting. All SQL functions always use the internal storage value of a field.

- **LENGTH** returns the length of the canonical form of a number. A number in canonical form excludes leading and trailing zeros, leading signs (except a single minus sign), and a trailing decimal separator character. **LENGTH** returns the string length of a numeric string. A numeric string is not converted to canonical form.

- **LENGTH** does not exclude leading blanks from strings. You can remove leading blanks from a string using the LTRIM function.

LENGTH differs from the other length functions (**$LENGTH**, **CHARACTER_LENGTH**, **CHAR_LENGTH**, and **DATALENGTH**) when performing the following operations:

- **LENGTH** excludes trailing blanks and the string-termination character.

  $LENGTH, **CHARACTER_LENGTH**, **CHAR_LENGTH**, and **DATALENGTH** do not exclude trailing blanks and terminators.

- **LENGTH** returns NULL if passed a NULL value, and 0 if passed an empty string.

  CHARACTER_LENGTH, **CHAR_LENGTH**, and **DATALENGTH** also return NULL if passed a NULL value, and 0 if passed an empty string. **$LENGTH** returns 0 if passed a NULL value, and 0 if passed an empty string.

- **LENGTH** does not support data stream fields. Specifying a stream field for *string-expression* results in an SQLCODE -37.

  $LENGTH also does not support stream fields. **CHARACTER_LENGTH**, **CHAR_LENGTH**, and **DATALENGTH** functions do support data stream fields.

Arguments

string-expression

A string expression, which can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR).

Examples

In the following example, InterSystems IRIS first converts each number to canonical form (removing leading and trailing zeros, resolving leading signs, and removing a trailing decimal separator character). Each **LENGTH** returns a length of 1:

SQL

```
SELECT {fn LENGTH(7.00)} AS CharCount,
       {fn LENGTH(+007)} AS CharCount,
       {fn LENGTH(007.)} AS CharCount,
       {fn LENGTH(00000.00)} AS CharCount,
       {fn LENGTH(-0)} AS CharCount
```

In the following example, the first **LENGTH** removes the leading zero, returning a length value of 2; the second **LENGTH** treats the numeric value as a string, and does not remove the leading zero, returning a length value of 3:

SQL

```
SELECT LENGTH(0.7) AS CharCount,
       LENGTH('0.7') AS CharCount
```

The following example returns the value 12:

SQL

```
SELECT LENGTH('INTERSYSTEMS') AS CharCount
```

The following example shows how **LENGTH** handles leading and trailing blanks. The first **LENGTH** returns 15, because **LENGTH** excludes trailing blanks, but not leading blanks. The second **LENGTH** returns 12, because **LTRIM** excludes the leading blanks:

SQL

```
SELECT LENGTH('   INTERSYSTEMS     ') AS CharCount,
       LENGTH(LTRIM('   INTERSYSTEMS     ')) AS CharCount
```

The following example returns the number of characters in each Name value in the Sample.Person table:

SQL

```
SELECT Name,{fn LENGTH(Name)} AS CharCount
FROM Sample.Person
ORDER BY CharCount
```

The following example returns the number of characters in the DOB (date of birth) field. Note that the length returned (by LENGTH, CHAR_LENGTH, and CHARACTER_LENGTH) is the internal ($HOROLOG) format of the date, not the display format. The display length of DOB is ten characters; all three length functions return the internal length of 5:

SQL

```
SELECT DOB,{fn LENGTH(DOB)} AS LenCount,
CHAR_LENGTH(DOB) AS CCount,
CHARACTER_LENGTH(DOB) AS CtrCount
FROM Sample.Person
```

The following Embedded SQL example gives the length of a string of Unicode characters. The length returned is the number of characters (7), not the number of bytes.

ObjectScript

```
SET a=$CHAR(920,913,923,913,931,931,913)
&sql(SELECT LENGTH(:a) INTO :b )
IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
ELSE {
    WRITE !,"The Greek Sea: ",a,!,$LENGTH(a),!,b }
```

See Also

- SQL functions: CHAR_LENGTH, CHARACTER_LENGTH, DATALENGTH, LEN, $LENGTH

- ObjectScript function: $LENGTH

$LENGTH (SQL)

A string function that returns the number of characters or the number of delimited substrings in a string.

Synopsis

```
$LENGTH(expression[,delimiter])
```

Description

$LENGTH returns the number of characters in a specified string or the number of substrings in a specified string, depending on the arguments used.

- **$LENGTH**(*expression*) returns the number of characters in the string. If the expression is an empty string ("), **$LENGTH** returns 0. If the expression is NULL, **$LENGTH** returns 0.

- **$LENGTH**(*expression,delimiter*) returns the number of substrings within the string. **$LENGTH** returns the number of substrings separated from one another by the indicated *delimiter*. This number is always equal to the number of delimiter instances found in the *expression* string, plus one.

$LENGTH(expression) and other Length Functions

$LENGTH(*expression*) and the other length functions (**LENGTH**, **CHARACTER_LENGTH**, **CHAR_LENGTH**, and **DATALENGTH**) all perform the following operations:

- **$LENGTH** returns the length of the Logical (internal data storage) value of a field, not the display value, regardless of the SelectMode setting. All SQL functions always use the internal storage value of a field.

- **$LENGTH** returns the length of the canonical form of a number. A number in canonical form excludes leading and trailing zeros, leading signs (except a single minus sign), and a trailing decimal separator character. **$LENGTH** returns the string length of a numeric string. A numeric string is not converted to canonical form.

- **$LENGTH** does not exclude leading blanks from strings. You can remove leading blanks from a string using the LTRIM function.

$LENGTH differs from the other length functions (**LENGTH**, **CHARACTER_LENGTH**, **CHAR_LENGTH**, and **DATALENGTH**) when performing the following operations:

- **$LENGTH** does not exclude trailing blanks and terminators.

  CHARACTER_LENGTH, **CHAR_LENGTH**, and **DATALENGTH** also do not exclude trailing blanks and terminators. **LENGTH** excludes trailing blanks and the string-termination character.

- **$LENGTH** returns 0 if passed a NULL value, and 0 if passed an empty string.

  LENGTH, **CHARACTER_LENGTH**, **CHAR_LENGTH**, and **DATALENGTH** return NULL if passed a NULL value, and 0 if passed an empty string.

- **$LENGTH** does not support data stream fields. Specifying a stream field for *string-expression* results in an SQLCODE -37.

  LENGTH also does not support stream fields. **CHARACTER_LENGTH**, **CHAR_LENGTH**, and **DATALENGTH** functions do support data stream fields.

- **$LENGTH** returns data type SMALLINT. All the other length functions return data type INTEGER.

NULL and Empty String Arguments

$LENGTH(*expression*) does not distinguish between the empty string (") and NULL (the absence of a value). It returns a length of 0 for both an empty string (") value and for NULL.

$LENGTH(*expression,delimiter*) with a non-null delimiter returns a delimited substring count of 1 if no match occurred. The full string is a single substring containing no delimiters. This is true even when *expression* is the empty string ("), or *expression* is NULL. However, an empty string does match itself, returning a value of 2.

The following table shows the possible combinations of a string ('abc'), empty string ("), or NULL *expression* value paired with a non-matching string ('^'), empty string ("), or NULL *delimiter* value:

$LENGTH(NULL) = 0	$LENGTH(") = 0	$LENGTH('abc') = 3
$LENGTH(NULL,NULL) = 0	$LENGTH(",NULL) = 0	$LENGTH('abc',NULL) = 0
$LENGTH(NULL,") = 1	$LENGTH(",") = 2	$LENGTH('abc',") = 1
$LENGTH(NULL,'^') = 1	$LENGTH(",'^') = 1	$LENGTH('abc','^') = 1

Arguments

expression

The target string. It can be a numeric value, a string literal, the name of any variable, or any valid expression.

delimiter

An optional string that demarcates separate substrings in the target string. It must be a string literal, but can be of any length. The enclosing quotation marks are required.

$LENGTH returns the SMALLINT data type.

Examples

The following example returns 6, the length of the string:

SQL

```
SELECT $LENGTH('ABCDEG') AS StringLength
```

The following example returns 3, the number of substrings within the string, as delimited by the dollar sign ($) character.

SQL

```
SELECT $LENGTH('ABC$DEF$EFG','$') AS SubStrings
```

If the specified delimiter is not found in the string **$LENGTH** returns 1, because the only substring is the string itself:

SQL

```
SELECT $LENGTH('ABCDEG','$') AS SubStrings
```

In the following example, the first **$LENGTH** function returns 11, the number of characters in *a* (including, of course, the space character). The second **$LENGTH** function returns 2, the number of substrings in *a* using *b*, the space character, as the substring delimiter.

SQL

```
SELECT $LENGTH("HELLO WORLD"), $LENGTH("HELLO WORLD"," ")
```

The following example returns 0 because the string tested is the null string:

SQL

```
SELECT $LENGTH(NULL) AS StringLength
```

The following example returns 1 because a delimiter is specified and not found. There is one substring, which is the null string:

SQL

```
SELECT $LENGTH(NULL,'$') AS SubStrings
```

The following example returns 0 because the delimiter is the null string:

SQL

```
SELECT $LENGTH('ABCDEFG',NULL) AS SubStrings
```

Notes

$LENGTH, $PIECE, and $LIST

- **$LENGTH** with one argument returns the number of characters in a string. This function can be used with the **$EXTRACT** function, which locates a substring by position and returns the substring value.

- **$LENGTH** with two arguments returns the number of substrings in a string, based on a delimiter. This function can be used with the **$PIECE** function, which locates a substring by a delimiter and returns the substring value.

- **$LENGTH** should not be used on encoded lists created using **$LISTBUILD** or **$LIST**. Use **$LISTLENGTH** to determine the number of substrings (list elements) in an encoded list string.

The **$LENGTH**, **$FIND**, **$EXTRACT**, and **$PIECE** functions operate on standard character strings. The various **$LIST** functions operate on encoded character strings, which are incompatible with standard character strings. The only exceptions are the **$LISTGET** function and the one-argument and two-argument forms of **$LIST**, which take an encoded character string as input, but output a single element value as a standard character string.

See Also

- SQL functions: CHAR_LENGTH, CHARACTER_LENGTH, DATALENGTH, $EXTRACT, $FIND, LENGTH, $LIST, $LISTGET, $PIECE

- ObjectScript functions: $EXTRACT, $FIND, $LENGTH, $LIST, $LISTBUILD, $LISTGET, $PIECE

$LIST (SQL)

A list function that returns elements in a list.

Synopsis

```
$LIST(list[,position[,end]])
```

Description

$LIST returns elements from a list. The elements returned depend on the arguments used.

- **$LIST**(*list*) returns the first element in the list as a text string.

- **$LIST**(*list,position*) returns the element indicated by the specified position as a text string, where a *position* value of 1 represents the first element in the list. The *position* argument must evaluate to an integer.

- **$LIST**(*list,position,end*) returns a "sublist" (an encoded list string) containing the elements of the list from the specified start *position* through the specified *end* position.

This function returns data of type VARCHAR.

Arguments

list

An encoded character string containing one or more elements. You can create a list using the SQL $LISTBUILD function or the ObjectScript $LISTBUILD function. You can convert a delimited string into a list using the SQL $LISTFROMSTRING function or the ObjectScript $LISTFROMSTRING function. You can extract a list from an existing list using the SQL $LIST function or the ObjectScript $LIST function.

position

The position of a list element to return. List elements are counted from 1. If *position* is omitted, the first element is returned. If the value of *position* is 0 or greater than the number of elements in the list, InterSystems SQL does not return a value. If the value of *position* is negative one (–1), **$LIST** returns the final element in the list. For example, the following command will return "Green":

SQL

```
SELECT $LIST($LISTBUILD("Red","Blue","Green"),-1)
```

If the *end* argument is specified, *position* specifies the first element in a range of elements. Even when only one element is returned (when *position* and *end* are the same number) this element is returned as an encoded list string. Thus, $LIST(x,2) (which returns the element as an ordinary string) is not identical to $LIST(x,2,2) (which returns the element as an encoded list string).

end

The position of the last element in a range of elements. You must specify *position* to specify *end*. When *end* is specified, the value returned is an encoded list string. Because of this encoding, such strings should only be processed by other $LIST functions.

If the value of *end* is:

- greater than *position*, an encoded string containing a list of elements is returned.

- equal to *position*, an encoded string containing the one element is returned.

- less than *position*, no value is returned.

- greater than the number of elements in *list*, it is equivalent to specifying the final element in the list.

- negative one (–1), it is equivalent to specifying the final element in the list.

When specifying *end*, you can specify a *position* value of zero (0). In this case, 0 is equivalent to 1.

Working with Lists

A table can contain one or more List fields. Because a list is an encoded string, these fields can be defined as data type %List (%Library.List) or data type VARCHAR. A field of data type %List can be identified as CType (client data type) = 6.

The data type does not restrict the field's permitted values upon insert or update. The user must therefore make sure that all data values in a List field are List encoded character strings. If the SQL **$LIST** function encounters a non-encoded string data value, the **SELECT** operation fails with an SQLCODE -400 alongside a %msg such as the following: Unexpected error occurred: <LIST>%0AmBuncommitted+1^%sqlcq.USER.cls61.1.

A list can be supplied to the SQL **$LIST** function by using a host variable, or by specifying a **$LISTBUILD** within SQL. Both are shown in the following Embedded SQL example:

ObjectScript

```
SET mylist=$LISTBUILD("Red","Blue","Green")
&sql(SELECT $LIST(:mylist,2),$LIST($LISTBUILD('Red','Blue','Green'),3)
INTO :a,:b )
IF SQLCODE'=0 {
  WRITE "Error code ",SQLCODE,! }
ELSE {
  WRITE !,"The host varable list element is ",a,!
  WRITE !,"The SQL $LISTBUILD list element is ",b,! }
```

A list can be extracted from another list by using the **$LIST** function:

SQL

```
SELECT $LIST($LISTBUILD("Red","Blue","Green"),2,3)
```

In the following Embedded SQL example, *sublist* is not a valid *list* argument, because it is a single element returned as an ordinary string, not an encoded list string. Only the three-argument form of **$LIST** returns an encoded list string. In this case, an SQLCODE -400 fatal error is generated:

ObjectScript

```
SET a=$LISTBUILD("Red","Blue","Green")
&sql(SELECT $LIST(:a,2)
INTO :sublist )
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
&sql(SELECT $LIST(:sublist,1)
INTO :c )
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"The sublist is"
  ZZDUMP c   ; Variable not set
  }
}
```

Examples

In the following example, both commands return "Red", the first element in the list. The first returns the first element by default, and the second returns the first element because the *position* argument is set to 1:

SQL

```
SELECT $LIST($LISTBUILD("Red","Blue","Green"))
SELECT $LIST($LISTBUILD("Red","Blue","Green"),1)
```

The following example returns "Blue", the second element in the list:

ObjectScript

```
SELECT $LIST($LISTBUILD("Red","Blue","Green"),2)
```

The following Embedded SQL example returns "Red Blue", a two-element list string beginning with the first element and ending with the second element in the list. **ZZDUMP** is used rather than **WRITE**, because a list string contains special (non-printing) encoding characters:

ObjectScript

```
    SET a=$LISTBUILD("Red","Blue","Green")
    &sql(SELECT $LIST(:a,1,2)
    INTO :b )
    IF SQLCODE'=0 {
      WRITE !,"Error code ",SQLCODE }
    ELSE {
      WRITE !,"The encoded sublist is"
      ZZDUMP b   ; Prints "Red Blue "
      }
```

The following example returns the last element in a list of unknown length. The first SELECT statement returns the last element as an ordinary string, whereas the second statement returns it as an encoded list string:

SQL

```
SELECT $LIST($LISTBUILD("Red","Blue","Green"),-1)
SELECT $LIST($LISTBUILD("Red","Blue","Green"),$LISTLENGTH($LISTBUILD("Red","Blue","Green")),-1)
```

Notes

Invalid Argument Values

If the expression in the *list* argument does not evaluate to a valid list, an SQLCODE -400 fatal error is generated:

SQL

```
SELECT $LIST("the quick brown fox",1)
```

If the value of the *position* argument or the *end* argument is less than -1, an SQLCODE -400 fatal error is generated:

ObjectScript

```
SELECT $LIST($LISTBUILD("Red","Blue","Green"),-2,3)
```

If the value of the *position* argument refers to a nonexistent list member and no *end* argument is used, an SQLCODE -400 fatal error is generated:

SQL

```
SELECT $LIST($LISTBUILD("Red","Blue","Green"),7)
```

However, if an *end* argument is used, no error occurs, and the null string is returned.

SQL

```
SELECT $LIST($LISTBUILD("Red","Blue","Green"),7,-1)
```

If the value of the *position* argument identifies an element with an undefined value, an SQLCODE –400 fatal error is generated:

SQL

```
SELECT $LIST($LISTBUILD("Red",,"Green"),2)
```

Two-Argument and Three-Argument $LIST

$LIST(*list*,1) is not equivalent to **$LIST**(*list*,1,1) because the former returns a string, while the latter returns a single-element list string. If there are no elements to return, the two-argument form does not return a value; the three-argument form returns a null string.

Unicode

If one Unicode character appears in a list element, that entire list element is represented as Unicode (wide) characters. Other elements in the list are not affected.

The following Embedded SQL example shows two lists. The *a* list consists of two elements which contain only ASCII characters. The *b* list consists of two elements: the first element contains a Unicode character (**$CHAR(960)** = the pi symbol); the second element contains only ASCII characters.

ObjectScript

```
SET a=$LISTBUILD("ABC"_$CHAR(68),"XYZ")
SET b=$LISTBUILD("ABC"_$CHAR(960),"XYZ")
&sql(SELECT $LIST(:a,1),$LIST(:a,2),$LIST(:b,1),$LIST(:b,2)
INTO :a1,:a2,:b1,:b2 )
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"The ASCII list a elements: "
  ZZDUMP a1
  ZZDUMP a2
  WRITE !,"The Unicode list b elements: "
  ZZDUMP b1
  ZZDUMP b2 }
```

Note that InterSystems IRIS encodes the first element of *b* entirely in wide Unicode characters. The second element of *b* contains no Unicode characters, and thus InterSystems IRIS encodes it using narrow ASCII characters.

See Also

- SQL functions: $LISTBUILD, $LISTDATA, $LISTFIND, $LISTFROMSTRING, $LISTGET, $LISTLENGTH, $LISTSAME, $LISTTOSTRING, $PIECE

- ObjectScript functions: $LIST, $LISTBUILD, $LISTDATA, $LISTFIND, $LISTFROMSTRING, $LISTGET, $LISTLENGTH, $LISTNEXT, $LISTSAME, $LISTTOSTRING, $LISTVALID

$LISTBUILD (SQL)

A list function that builds a list from strings.

Synopsis

```
$LISTBUILD(element [,...])
```

Description

$LISTBUILD takes one or more expressions and returns a list with one element for each expression.

The following functions can be used to create a list:

- **$LISTBUILD**, which creates a list from multiple strings, one string per element.

- **$LISTFROMSTRING**, which creates a list from a single string containing multiple delimited elements.

- **$LIST**, which extracts a sublist from an existing list.

$LISTBUILD is used with the other InterSystems SQL list functions: **$LIST**, **$LISTDATA**, **$LISTFIND**, **$LISTFROMSTRING**, **$LISTGET**, **$LISTLENGTH**, and **$LISTTOSTRING**.

Note: **$LISTBUILD** and the other $LIST functions use an optimized binary representation to store data elements. For this reason, equivalency tests may not work as expected with some $LIST data. Data that might, in other contexts, be considered equivalent, may have a different internal representation. For example, `$LISTBUILD(1)` is not equal to `$LISTBUILD('1')`.

For the same reason, a list string value returned by **$LISTBUILD** should not be used in character search and parse functions that use a delimiter character, such as **$PIECE** and the two-argument form of **$LENGTH**. Elements in a list created by **$LISTBUILD** are not marked by a character delimiter, and thus can contain any character.

Arguments

element

Any expression, or comma-separated list of expressions.

Examples

The following example takes three strings and produces a three-element list:

SQL

```
SELECT $LISTBUILD("Red","White","Blue")
```

Notes

Omitting Arguments

Omitting an element expression yields an element whose value is NULL. For example, the following example contains two **$LISTBUILD** statements that both produce a three-element list whose second element has an undefined (NULL) value:

SQL

```
SELECT $LISTBUILD("Red",,"Blue"), $LISTBUILD("Red",'',"Blue")
```

Additionally, if a **$LISTBUILD** expression is undefined, the corresponding list element has an undefined value. The following example produces a two-element list whose first element is "Red" and whose second element has an undefined value:

SQL

```
SELECT $LISTBUILD('Red',:z)
```

The following example produces a two-element list. The trailing comma indicates the second element has an undefined value:

SQL

```
SELECT $LISTBUILD('Red',)
```

Providing No Arguments

Invoking the **$LISTBUILD** function with no arguments returns a list with one element whose data value is undefined. This is not the same as NULL. The following are valid **$LISTBUILD** statements that create "empty" lists:

SQL

```
SELECT $LISTBUILD(), $LISTBUILD(NULL)
```

The following are valid **$LISTBUILD** statements that create a list element that contains an empty string:

SQL

```
SELECT $LISTBUILD(''), $LISTBUILD(CHAR(0))
```

Nesting Lists

An element of a list may itself be a list. For example, the following statement produces a three-element list whose third element is the two-element list, "Walnut,Pecan":

SQL

```
SELECT $LISTBUILD('Apple','Pear',$LISTBUILD('Walnut','Pecan'))
```

Concatenating Lists

The result of concatenating two lists with the SQL Concatenate operator (‖) is another list. For example, the following **SELECT** items produce the same list, "A,B,C":

SQL

```
SELECT $LISTBUILD('A','B','C') AS List,
  $LISTBUILD('A','B')||$LISTBUILD('C') AS CatList
```

In the following example, the first two select items result in the same two-element list; the third select item results in NULL (because concatenating NULL to anything results in NULL); the fourth and fifth select items result in the same three-element list:

SQL

```
SELECT
  $LISTBUILD('A','B') AS List,
  $LISTBUILD('A','B')||'' AS CatEStr,
  $LISTBUILD('A','B')||NULL AS CatNull,
  $LISTBUILD('A','B')||$LISTBUILD('') AS CatEList,
  $LISTBUILD('A','B')||$LISTBUILD(NULL) AS CatNList
```

Unicode

If one or more characters in a list element is a wide (Unicode) character, all characters in that element are represented as wide characters. To ensure compatibility across systems, **$LISTBUILD** always stores these bytes in the same order, regardless of the hardware platform. Wide characters are represented as byte strings. For further details, refer to the ObjectScript $LISTBUILD function.

See Also

- SQL functions: $LIST, $LISTDATA, $LISTFIND, $LISTFROMSTRING, $LISTGET, $LISTLENGTH, $LISTSAME, $LISTTOSTRING, $PIECE

- ObjectScript functions: $LIST, $LISTBUILD, $LISTDATA, $LISTFIND, $LISTFROMSTRING, $LISTGET, $LISTLENGTH, $LISTNEXT, $LISTSAME, $LISTTOSTRING, $LISTVALID

$LISTDATA (SQL)

A list function that indicates whether the specified element exists and has a data value.

Synopsis

```
$LISTDATA(list[,position])
```

Description

$LISTDATA checks for data in the requested element in a list. **$LISTDATA** returns a value of 1 if the element indicated by the *position* argument is in the *list* and has a data value. **$LISTDATA** returns a value of a 0 if the element is not in the *list* or does not have a data value.

This function returns data of type SMALLINT.

Arguments

list

An encoded character string containing one or more elements. You can create a list using the SQL $LISTBUILD function or the ObjectScript $LISTBUILD function. You can convert a delimited string into a list using the SQL $LISTFROMSTRING function or the ObjectScript $LISTFROMSTRING function. You can extract a list from an existing list using the SQL $LIST function or the ObjectScript $LIST function.

position

If you omit the *position* argument, **$LISTDATA** evaluates the first element. If the value of the *position* argument is -1, it is equivalent to specifying the final element of the list. If the value of the *position* argument refers to a nonexistent list member, **$LISTDATA** returns 0.

Examples

The following examples show the results of the various values of the *position* argument.

All of the following **$LISTDATA** statements return a value of 1:

SQL

```
SELECT
    $LISTDATA($LISTBUILD("Red",,Y,"","Green")),
    $LISTDATA($LISTBUILD("Red",,Y,"","Green"),1),
    $LISTDATA($LISTBUILD("Red",,Y,"","Green"),4),
    $LISTDATA($LISTBUILD("Red",,Y,"","Green"),5),
    $LISTDATA($LISTBUILD("Red",,Y,"","Green"),-1)
```

The following **$LISTDATA** statements return a value of 0 for the same five-element list:

SQL

```
SELECT
    $LISTDATA($LISTBUILD("Red",,Y,"","Green"),2),
    $LISTDATA($LISTBUILD("Red",,Y,"","Green"),3),
    $LISTDATA($LISTBUILD("Red",,Y,"","Green"),0),
    $LISTDATA($LISTBUILD("Red",,Y,"","Green"),6)
```

Notes

Invalid Argument Values

If the expression in the *list* argument does not evaluate to a valid list, an SQLCODE -400 fatal error occurs:

SQL

```
SELECT $LISTDATA('fred')
```

If the value of the *position* argument is less than -1, an SQLCODE -400 fatal error occurs:

SQL

```
SELECT $LISTDATA($LISTBUILD("Red","Blue","Green"),-3)
```

This does not occur when *position* is a nonnumeric value:

SQL

```
SELECT $LISTDATA($LISTBUILD("Red","Blue","Green"),'g')
```

See Also

- SQL functions: $LIST, $LISTBUILD, $LISTFIND, $LISTFROMSTRING, $LISTGET, $LISTLENGTH, $LISTSAME, $LISTTOSTRING, $PIECE

- ObjectScript functions: $LIST, $LISTBUILD, $LISTDATA, $LISTFIND, $LISTFROMSTRING, $LISTGET, $LISTLENGTH, $LISTNEXT, $LISTSAME, $LISTTOSTRING, $LISTVALID

$LISTFIND (SQL)

A list function that searches a specified list for the requested value.

Synopsis

```
$LISTFIND(list,value[,startafter])
```

Description

$LISTFIND searches the specified *list* for the first instance of the requested *value*. The search begins with the element after the position indicated by the *startafter* argument. If you omit the *startafter* argument, **$LISTFIND** assumes a *startafter* value of 0 and starts the search with the first element (element 1). If the value is found, **$LISTFIND** returns the position of the matching element. If the value is not found or if the value of the *startafter* argument refers to a nonexistent list member, **$LISTFIND** returns a 0.

This function returns data of type SMALLINT.

Arguments

list

An expression that evaluates to a valid list. A *list* is an encoded character string containing one or more elements. You can create a *list* using the SQL or ObjectScript **$LISTBUILD** or **$LISTFROMSTRING** functions. You can extract a *list* from an existing list using the SQL or ObjectScript **$LIST** function.

value

An expression containing the search element. A character string.

startafter

An optional integer expression interpreted as a list position. The search starts with the element after this position. Zero and –1 are valid values; –1 never returns an element. Zero is the default.

Examples

The following example returns 2, the position of the first occurrence of the requested string:

SQL

```
SELECT $LISTFIND($LISTBUILD("Red","Blue","Green"),'Blue')
```

The following example returns 0, indicating the requested string was not found:

ObjectScript

```
SELECT $LISTFIND($LISTBUILD("Red","Blue","Green"),'Orange')
```

The following three examples show the effect of using the *startafter* argument. The first example does not find the requested string and returns 0 because the requested string occurs at the *startafter* position:

SQL

```
SELECT $LISTFIND($LISTBUILD("Red","Blue","Green"),'Blue',2)
```

The second example finds the requested string at the first position by setting *startafter* to zero (the default value):

SQL

```
SELECT $LISTFIND($LISTBUILD("Red","Blue","Green"),'Red',0)
```

The third example finds the second occurrence of the requested string and returns 5, because the first occurs before the *startafter* position:

SQL

```
SELECT $LISTFIND($LISTBUILD("Red","Blue","Green","Yellow","Blue"),'Blue',3)
```

The **$LISTFIND** function only matches complete elements. Thus, the following example returns 0 because no element of the list is equal to the string "B", though all of the elements contain "B":

ObjectScript

```
SELECT $LISTFIND($LISTBUILD("ABC","BCD","BBB"),'B')
```

Notes

Invalid Argument Values

If the expression in the *list* argument does not evaluate to a valid list, the **$LISTFIND** function generates an SQLCODE -400 fatal error.

SQL

```
SELECT $LISTFIND("Blue",'Blue')
```

If the value of the *startafter* argument is -1, **$LISTFIND** always returns zero (0).

SQL

```
SELECT $LISTFIND($LISTBUILD("Red","Blue","Green"),'Blue',-1)
```

If the value of the *startafter* argument is less than -1, invoking the **$LISTFIND** function generates an SQLCODE -400 fatal error.

ObjectScript

```
SELECT $LISTFIND($LISTBUILD("Red","Blue","Green"),'Blue',-3)
```

See Also

- SQL functions: $LIST, $LISTBUILD, $LISTDATA, $LISTFROMSTRING, $LISTGET, $LISTLENGTH, $LISTSAME, $LISTTOSTRING, $PIECE

- ObjectScript functions: $LIST, $LISTBUILD, $LISTDATA, $LISTFIND, $LISTFROMSTRING, $LISTGET, $LISTLENGTH, $LISTNEXT, $LISTSAME, $LISTTOSTRING, $LISTVALID

$LISTFROMSTRING (SQL)

A list function that creates a list from a string.

Synopsis

```
$LISTFROMSTRING(string[,delimiter])
```

Description

$LISTFROMSTRING takes a quoted string containing delimited elements and returns a list. A list represents data in an encoded format which does not use delimiter characters. Thus a list can contain all possible characters, and is ideally suited for bitstring data. Lists are handled using the ObjectScript and InterSystems SQL **$LIST** functions.

Arguments

string

A string literal (enclosed in single quotation marks), a numeric, or a variable or expression that evaluates to a string. This string can contain one or more substrings (elements), separated by a *delimiter*. The string data elements must not contain the *delimiter* character (or string), because the *delimiter* character is not included in the output list.

delimiter

An optional character (or string of characters) used to delimit substrings within the input string. It can be a numeric or string literal (enclosed in single quotation marks), the name of a variable, or an expression that evaluates to a string.

Commonly, a delimiter is a designated character which is never used within string data, but is set aside solely for use as a delimiter separating substrings. A delimiter can also be a multi-character string, the individual characters of which can be used within string data. If you specify no delimiter, the default delimiter is the comma (,) character.

Examples

The following example takes a string of names which are separated by a blank space, and creates a list:

SQL

```
SELECT $LISTFROMSTRING("Deborah Noah Martha Bowie",' ')
```

The following example uses the default delimiter (the comma character), and creates a list:

SQL

```
SELECT $LISTFROMSTRING("Deborah,Noah,Martha,Bowie")
```

See Also

- SQL functions: $LIST $LISTBUILD $LISTDATA $LISTFIND $LISTGET $LISTLENGTH $LISTSAME $LIST-TOSTRING $PIECE

- ObjectScript functions: $LIST $LISTBUILD $LISTDATA $LISTFIND $LISTFROMSTRING $LISTGET $LISTLENGTH $LISTNEXT $LISTSAME $LISTTOSTRING $LISTVALID

$LISTGET (SQL)

A list function that returns an element in a list or a specified default value.

Synopsis

```
$LISTGET(list[,position[,default]])
```

Description

$LISTGET returns the requested element in the specified list as a standard character string. If the value of the *position* argument refers to a nonexistent member or identifies an element with an undefined value, the specified default value is returned.

The **$LISTGET** function is identical to the one- and two-argument forms of the **$LIST** function except that, under conditions that would cause **$LIST** to return a null string, **$LISTGET** returns a default value.

This function returns data of type VARCHAR.

You can use **$LISTGET** to retrieve a field value from a serial container field. In the following example, Home is a serial container field, the third element of which is Home_State:

SQL

```
SELECT Name,$LISTGET(Home,3) AS HomeState
FROM Sample.Person
```

Arguments

list

An encoded character string containing one or more elements. You can create a list using the SQL $LISTBUILD function or the ObjectScript $LISTBUILD function. You can convert a delimited string into a list using the SQL $LISTFROMSTRING function or the ObjectScript $LISTFROMSTRING function. You can extract a list from an existing list using the SQL $LIST function or the ObjectScript $LIST function.

position

The *position* argument must evaluate to an integer. If it is omitted, by default, the function examines the first element of the list. If the value of the *position* argument is -1, it is equivalent to specifying the last element of the list.

default

A character string. If you omit the *default* argument, a zero-length string is assumed for the default value.

Examples

The **$LISTGET** functions in the following example both return "Red", the first element in the list:

SQL

```
SELECT
    $LISTGET($LISTBUILD("Red","Blue","Green")),
    $LISTGET($LISTBUILD("Red","Blue","Green"),1)
```

The **$LISTGET** functions in the following example both return "Green", the third and last element in the list:

ObjectScript

```
SELECT
    $LISTGET($LISTBUILD("Red","Blue","Green"),3),
    $LISTGET($LISTBUILD("Red","Blue","Green"),-1)
```

The **$LISTGET** functions in the following example both return a value upon encountering the undefined 2nd element in the list. The first returns a question mark (?), which the user defined as the default value. The second returns a null string because a default value is not specified:

SQL

```
SELECT
    $LISTGET($LISTBUILD("Red",,"Green"),2,'?'),
    $LISTGET($LISTBUILD("Red",,"Green"),2)
```

The **$LISTGET** functions in the following example both specify a position greater than the last element in the three-element list. The first returns a null string because the default value is not specified. The second returns the user-specified default value, "ERR":

SQL

```
SELECT
    $LISTGET($LISTBUILD("Red","Blue","Green"),4),
    $LISTGET($LISTBUILD("Red","Blue","Green"),4,'ERR')
```

The **$LISTGET** functions in the following example both return a null string:

SQL

```
SELECT
    $LISTGET($LISTBUILD("Red","Blue","Green"),0),
    $LISTGET(NULL)
```

Notes

Invalid Argument Values

If the expression in the *list* argument does not evaluate to a valid list, an SQLCODE -400 fatal error occurs because the **$LISTGET** return variable remains undefined. This occurs even when a *default* value is supplied, as in the following example:

SQL

```
SELECT $LISTGET('fred',1,'failsafe')
```

If the value of the *position* argument is less than -1, an SQLCODE -400 fatal error occurs because the **$LISTGET** return variable remains undefined. This occurs even when a *default* value is supplied, as in the following example:

SQL

```
SELECT $LISTGET($LISTBUILD("Red","Blue","Green"),-3,'failsafe')
```

This does not occur when *position* is a nonnumeric value:

SQL

```
SELECT $LISTGET($LISTBUILD("Red","Blue","Green"),'g','failsafe')
```

See Also

- SQL functions: $LIST $LISTBUILD $LISTDATA $LISTFIND $LISTFROMSTRING $LISTLENGTH $LISTSAME $LISTTOSTRING $PIECE

- ObjectScript functions: $LIST $LISTBUILD $LISTDATA $LISTFIND $LISTFROMSTRING $LISTGET $LISTLENGTH $LISTNEXT $LISTSAME $LISTTOSTRING $LISTVALID

$LISTLENGTH (SQL)

A list function that returns the number of elements in a specified list.

Synopsis

```
$LISTLENGTH(list)
```

Description

$LISTLENGTH returns the number of elements in *list*.

This function returns data of type SMALLINT.

Arguments

list

An expression that evaluates to a valid list. A *list* is an encoded character string containing one or more elements. You can create a *list* using the SQL or ObjectScript **$LISTBUILD** or **$LISTFROMSTRING** functions. You can extract a *list* from an existing list using the SQL or ObjectScript **$LIST** function.

Examples

The following example returns 3, because there are 3 elements in the list:

SQL

```
SELECT $LISTLENGTH($LISTBUILD("Red","Blue","Green"))
```

The following SQL example also returns 3, because there are 3 elements in the list:

SQL

```
SELECT $LISTLENGTH($LISTBUILD('Red','Blue','Green'))
```

The following example also returns 3. There are 3 elements in the list, though the second element contains no data:

SQL

```
SELECT $LISTLENGTH($LISTBUILD("Red",,"Green"))
```

In the following SQL example, each **$LISTLENGTH** returns 3, because there are 3 elements in the list, though the second element contains no data:

SQL

```
SELECT $LISTLENGTH($LISTBUILD('Red','','Green')),
       $LISTLENGTH($LISTBUILD('Red',NULL,'Green')),
       $LISTLENGTH($LISTBUILD('Red',,'Green'))
```

Notes

Invalid Lists

If *list* is not a valid list, an SQLCODE -400 fatal error is generated:

SQL

```
SELECT $LISTLENGTH("fred")
```

If the ObjectScript **$LISTBUILD** function is used to build a list that contains only the null string, this is a valid *list*, containing one element:

ObjectScript

```
SELECT $LISTLENGTH($LISTBUILD(""))
```

Null Lists

The SQL **$LISTLENGTH** function and the ObjectScript **$LISTLENGTH** function differ in how they handle a null list (a list containing no elements).

The following three examples show how the **$LISTLENGTH** SQL function handles a null list. In the first two examples, *list* is the null string, and a null string is returned:

SQL

```
SELECT $LISTLENGTH("")
```

SQL

```
SELECT $LISTLENGTH(NULL)
```

In the third example, *list* is the value $CHAR(0), which is an invalid list; an SQLCODE -400 fatal error is generated:

SQL

```
SELECT $LISTLENGTH('')
```

Note that this differs from how the ObjectScript **$LISTLENGTH** function handles a null list. In ObjectScript, the null string ("") is used to represent a null list, a list containing no elements. Because it contains no list elements, it has a **$LISTLENGTH** count of 0, as shown in the following example:

ObjectScript

```
   WRITE $LISTLENGTH("")
```

$LISTLENGTH and Nested Lists

The following example returns 3, because **$LISTLENGTH** does not recognize the individual elements in nested lists:

ObjectScript

```
SELECT $LISTLENGTH($LISTBUILD("Apple","Pear",$LISTBUILD("Walnut","Pecan")))
```

See Also

- SQL list functions: $LIST, $LISTBUILD, $LISTDATA, $LISTFIND, $LISTFROMSTRING, $LISTGET, $LISTSAME, $LISTTOSTRING

- Other SQL functions: $PIECE

- ObjectScript list functions: $LIST, $LISTBUILD, $LISTDATA, $LISTFIND, $LISTFROMSTRING, $LISTGET, $LISTLENGTH, $LISTNEXT, $LISTSAME, $LISTTOSTRING, $LISTVALID

$LISTSAME (SQL)

A list function that compares two lists and returns a boolean value.

Synopsis

```
$LISTSAME(list1,list2)
```

Description

$LISTSAME compares the contents of two lists and returns 1 if the lists are the same. If the lists are not the same, **$LISTSAME** returns 0. **$LISTSAME** compares the two lists element-by-element. For two lists to be the same, they must contain the same number of elements and each element in *list1* must match the corresponding element in *list2*.

$LISTSAME compares list elements using their string representations. **$LISTSAME** comparisons are case-sensitive. **$LISTSAME** compares the two lists element-by-element in left-to-right order. Therefore **$LISTSAME** returns a value of 0 when it encounters the first non-matching pair of list elements; it does not check subsequent items to determine if they are valid list elements.

This function returns data of type SMALLINT.

Arguments

list (list1 and list2)

A *list* is an encoded character string containing one or more elements. You can create a list using the SQL $LISTBUILD function or the ObjectScript $LISTBUILD function. You can convert a delimited string into a list using the SQL $LIST-FROMSTRING function or the ObjectScript $LISTFROMSTRING function. You can extract a list from an existing list using the SQL $LIST function or the ObjectScript $LIST function.

The following are examples of valid lists:

- `$LISTBUILD('a','b','c')`: a three-element list.

- `$LISTBUILD('a','','c')`: a three-element list, the second element of which has a null string value.

- `$LISTBUILD('a',,'c')` or `$LISTBUILD('a',NULL,'c')`: a three-element list, the second element of which has no value.

- `$LISTBUILD(NULL,NULL)` or `$LISTBUILD(,NULL)`: a two-element list, the elements of which have no values.

- `$LISTBUILD(NULL)` or `$LISTBUILD()`: a one-element list, the element has no value.

If a *list* argument is NULL, **$LISTSAME** returns NULL. If a *list* argument is not a valid list (and is not NULL), InterSystems SQL generates an SQLCODE -400 fatal error.

Examples

The following example uses **$LISTSAME** to compare two list arguments:

SQL

```
SELECT $LISTSAME($LISTBUILD("Red",,"Yellow","Green","","Violet"),
$LISTBUILD("Red",,"Yellow","Green","","Violet"))
```

The following SQL example compares lists with NULL, absent, or empty string elements:

SQL

```
SELECT $LISTSAME($LISTBUILD('Red',NULL,'Blue'),$LISTBUILD('Red',,'Blue')) AS NullAbsent,
       $LISTSAME($LISTBUILD('Red',NULL,'Blue'),$LISTBUILD('Red','','Blue')) AS NullEmpty,
       $LISTSAME($LISTBUILD('Red',,'Blue'),$LISTBUILD('Red','','Blue')) AS AbsentEmpty
```

$LISTSAME comparison is not the same equivalence test as the one used by the ObjectScript equal sign. An equal sign compares the two lists as encoded strings (character-by-character); **$LISTSAME** compares the two lists element-by-element. This distinction is easily seen when comparing a number and a numeric string, as in the following example:

ObjectScript

```
  SET a = $LISTBUILD("365")
  SET b = $LISTBUILD(365)
  IF a=b
    { WRITE "Equal sign: lists a and b are the same",! }
  ELSE { WRITE "Equal sign: lists a and b are not the same",! }
&sql(SELECT $LISTSAME(:a,:b)
     INTO :c )
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSEIF c=1 { WRITE "$LISTSAME: lists a and b are the same",! }
ELSE { WRITE "$LISTSAME: lists a and b are not the same",! }
```

The following SQL example compares lists containing numbers and numeric strings in canonical and non-canonical forms. When comparing a numeric list element and a string list element, the string list element must represent the numeric in canonical form; this is because InterSystems IRIS always reduces numbers to canonical form before performing a comparison. In the following example, **$LISTSAME** compares a string and a number. The first three **$LISTSAME** functions return 1 (identical); the fourth **$LISTSAME** function returns 0 (not identical) because the string representation is not in canonical form:

SQL

```
SELECT $LISTSAME($LISTBUILD('365'),$LISTBUILD(365)),
       $LISTSAME($LISTBUILD('365'),$LISTBUILD(365.0)),
       $LISTSAME($LISTBUILD('365.5'),$LISTBUILD(365.5)),
       $LISTSAME($LISTBUILD('365.0'),$LISTBUILD(365.0))
```

See Also

- SQL functions: $LIST $LISTBUILD $LISTDATA $LISTFIND $LISTFROMSTRING $LISTGET $LISTLENGTH $LISTTOSTRING $PIECE

- ObjectScript functions: $LIST $LISTBUILD $LISTDATA $LISTFIND $LISTFROMSTRING $LISTGET $LISTLENGTH $LISTNEXT $LISTSAME $LISTTOSTRING $LISTVALID

$LISTTOSTRING (SQL)

A list function that creates a string from a list.

Synopsis

```
$LISTTOSTRING(list[,delimiter])
```

Description

$LISTTOSTRING takes an InterSystems IRIS list and converts it to a string. In the resulting string, the elements of the list are separated by the *delimiter*.

A list represents data in an encoded format which does not use delimiter characters. Thus a list can contain all possible characters, and is ideally suited for bitstring data. **$LISTTOSTRING** converts this list to a string with delimited elements. It sets aside a specified character (or character string) to serve as a delimiter. These delimited elements can be handled using the **$PIECE** function.

Note: The *delimiter* specified here must not occur in the source data. InterSystems IRIS makes no distinction between a character serving as a delimiter and the same character as a data character.

You can use **$LISTTOSTRING** to retrieve field values from a serial container field as a delimited string. In the following example, Home is a serial container field. It contains the list elements Home_Street, Home_City, Home_State, and Home_Zip:

SQL

```
SELECT Name,$LISTTOSTRING(Home,'^') AS HomeAddress
FROM Sample.Person
```

Arguments

list

An encoded character string containing one or more elements. You can create a list using the SQL $LISTBUILD function or the ObjectScript $LISTBUILD function. You can convert a delimited string into a list using the SQL $LISTFROMSTRING function or the ObjectScript $LISTFROMSTRING function. You can extract a list from an existing list using the SQL $LIST function or the ObjectScript $LIST function.

If the expression in the *list* argument does not evaluate to a valid list, an SQLCODE -400 error occurs.

delimiter

An optional character (or string of characters) used to delimit substrings within the output string. It can be a numeric or string literal (enclosed in single quotation marks), a host variable, or an expression that evaluates to a string.

Commonly, a delimiter is a designated character which is never used within string data, but is set aside solely for use as a delimiter separating substrings. A delimiter can also be a multi-character string, the individual characters of which can be used within string data.

If you specify no delimiter, the default delimiter is the comma (,) character. You can specify a null string ('') as a delimiter; in this case, substrings are concatenated with no delimiter. To specify the single quote character as the delimiter, duplicate the quote character thus: `''''` — four single quote characters.

Example

The following example converts the values of a list field to a string with the elements delimited by the colon (:) character:

SQL

```
SELECT
Name,
FavoriteColors AS ColorList,
$LISTTOSTRING(FavoriteColors,':') AS ColorStrings
FROM Sample.Person
WHERE FavoriteColors IS NOT NULL
```

See Also

- SQL functions: $LIST $LISTBUILD $LISTDATA $LISTFIND $LISTFROMSTRING $LISTGET $LISTLENGTH $LISTSAME $PIECE

- ObjectScript functions: $LIST $LISTBUILD $LISTDATA $LISTFIND $LISTFROMSTRING $LISTGET $LISTLENGTH $LISTNEXT $LISTSAME $LISTTOSTRING $LISTVALID

LOG (SQL)

A scalar numeric function that returns the natural logarithm of a given numeric expression.

Synopsis

```
{fn LOG(expression)}
```

Description

LOG returns the natural logarithm (base e) of *expression*. **LOG** returns a value with a precision of 21 and a scale of 18.

LOG can only be used as an ODBC scalar function (with the curly brace syntax).

Arguments

expression

A numeric expression.

LOG returns either the NUMERIC or DOUBLE data type. If *expression* is data type DOUBLE, **LOG** returns DOUBLE; otherwise, it returns NUMERIC.

Examples

The following example returns the natural logarithm of an integer:

SQL

```
SELECT {fn LOG(5)} AS Logarithm
```

returns 1.60943791...

The following Embedded SQL example shows the relationship between the **LOG** and **EXP** functions for the integers 1 through 10:

ObjectScript

```
   SET a=1
   WHILE a<11 {
   &sql(SELECT {fn LOG(:a)} INTO :b)
   IF SQLCODE'=0 {
     WRITE !,"Error code ",SQLCODE
     QUIT }
   ELSE {
     WRITE !,"Logarithm of ",a," = ",b }
   &sql(SELECT ROUND({fn EXP(:b)},12) INTO :c)
   IF SQLCODE'=0 {
     WRITE !,"Error code ",SQLCODE
     QUIT }
   ELSE {
     WRITE !,"Exponential of log ",b," = ",c
     SET a=a+1 }
   }
```

Note that the **ROUND** function is needed here to correct for very small discrepancies caused by system calculation limitations. In the above example, **ROUND** is set arbitrarily to 12 decimal digits for this purpose.

See Also

- SQL functions: EXP, LOG10, ROUND

- ObjectScript function: $ZLN

LOG10 (SQL)

A scalar numeric function that returns the base-10 logarithm of a given numeric expression.

Synopsis

```
{fn LOG10(expression)}
```

Description

LOG10 returns the base-10 logarithm value of *expression*. **LOG10** returns a value with a precision of 21 and a scale of 18.

LOG10 can only be used as an ODBC scalar function (with the curly brace syntax).

Arguments

expression

A numeric expression.

LOG10 returns either the NUMERIC or DOUBLE data type. If *expression* is data type DOUBLE, **LOG10** returns DOUBLE; otherwise, it returns NUMERIC.

Examples

The following example returns the base-10 logarithm of an integer:

SQL

```
SELECT {fn LOG10(5)} AS Log10
```

returns .69897000433...

The following Embedded SQL example returns the base-10 logarithm values for the integers 1 through 10:

ObjectScript

```
SET a=1
WHILE a<11 {
&sql(SELECT {fn LOG10(:a)} INTO :b)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE
  QUIT }
ELSE {
  WRITE !,"Log-10 of ",a," = ",b
  SET a=a+1 }
}
```

See Also

- SQL functions: EXP, LOG, ROUND

- ObjectScript function: $ZLOG

LOWER (SQL)

A case-transformation function that converts all uppercase letters in a string expression to lowercase letters.

Synopsis

```
LOWER(string-expression)
```

Description

The **LOWER** function converts uppercase letters to lowercase for display purposes. This is the inverse of the **UPPER** function. **LOWER** has no effects on non-alphabetic characters. It leave unchanged punctuation, numbers, and leading and trailing blank spaces.

LOWER does not force a numeric to be interpreted as a string. InterSystems SQL converts numerics to canonical form, removing leading and trailing zeros. A numeric specified as a string is not converted to canonical form, and retains leading and trailing zeros.

The **LCASE** function can also be used convert uppercase letters to lowercase.

LOWER has no effect on collation. The **%SQLUPPER** function is the preferred way in SQL to convert a data value for not case-sensitive collation. Refer to %SQLUPPER for further information on case transformation for collation.

Arguments

string-expression

The string expression whose characters are to be converted to lowercase. The expression can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR).

Examples

The following example returns each person's name in lowercase letters:

SQL

```
SELECT Name,LOWER(Name) AS LowName
    FROM Sample.Person
```

LOWER also works on Unicode (non-ASCII) alphabetic characters, as shown in the following example, which converts Greek letters from uppercase to lowercase:

SQL

```
SELECT LOWER($CHAR(920,913,923,913,931,931,913))
    FROM Sample.Person
```

See Also

- SQL functions: LCASE, UCASE

- ObjectScript function: $ZCONVERT

LPAD (SQL)

A string function that returns a string left-padded to a specified length.

Synopsis

```
LPAD(string-expression,length[,padstring])
```

Description

LPAD pads a string expression with leading pad characters. It returns a copy of the string padded to *length* number of characters. If the string expression is longer than *length* number of characters, the return string is truncated to *length* number of characters.

If *string-expression* is NULL, **LPAD** returns NULL. If *string-expression* is the empty string (") **LPAD** returns a string consisting entirely of pad characters. The returned string is type VARCHAR.

LPAD can be used in queries against a linked table.

LPAD does not remove leading or trailing blanks; it pads the string including any leading or trailing blanks. To remove leading or trailing blanks before padding a string, use **LTRIM**, **RTRIM**, or **TRIM**.

LPAD and $JUSTIFY

The two-argument form of **LPAD** and the two-argument form of **$JUSTIFY** both right-align a string by padding it with leading spaces. These two-argument forms differ in how they handle an output *length* that is shorter than the length of the input *string-expression*: **LPAD** truncates the input string to fit the specified output length while **$JUSTIFY** expands the output length to fit the input string. This is shown in the following example:

SQL

```
SELECT '>'||LPAD(12345,10)||'<' AS lpadplus,
       '>'||$JUSTIFY(12345,10)||'<' AS justifyplus,
       '>'||LPAD(12345,3)||'<' AS lpadminus,
       '>'||$JUSTIFY(12345,3)||'<' AS justifyminus
```

Arguments

string-expression

A string expression, which can be the name of a column, a string literal, a host variable, or the result of another scalar function. Can be of any data type convertible to a VARCHAR data type. *string-expression* cannot be a stream.

length

An integer specifying the number of characters in the returned string.

padstring

An optional string consisting of a character or a string of characters used to pad the input *string-expression*. The *padstring* character or characters are appended to the left of *string-expression* to supply as many characters as need to create an output string of *length* characters. *padstring* may be a string literal, a column, a host variable, or the result of another scalar function. If omitted, the default is a blank space character.

Examples

The following example left pads column values with ^ characters (when needed) to return strings of length 16. Note that some Name strings are left padded, some Name strings are right truncated to return strings of length 16.

SQL

```
SELECT TOP 15 Name,LPAD(Name,16,'^') AS Name16
FROM Sample.Person
```

The following example left pads column values with the ^=^ pad string (when needed) to return strings of length 20. Note that the pad name string is repeated as many times as needed, and that some return strings contain partial pad strings:

SQL

```
SELECT TOP 15 Name,LPAD(Name,20,'^=^') AS Name20
FROM Sample.Person
```

See Also

- $JUSTIFY function

- RPAD function

- LTRIM function

- RTRIM function

- TRIM function

LTRIM (SQL)

A string function that returns a string with the leading blanks removed.

Synopsis

```
LTRIM(string-expression)
```

```
{fn LTRIM(string-expression)}
```

Description

LTRIM removes the leading blanks from a string expression, and returns the string as type VARCHAR. If *string-expression* is NULL, **LTRIM** returns NULL. If *string-expression* is a string consisting entirely of blank spaces, **LTRIM** returns the empty string ('').

LTRIM leaves trailing blanks; to remove trailing blanks, use **RTRIM**. To remove leading and/or trailing characters of any type, use **TRIM**. To pad a string with leading blanks or other characters, use **LPAD**. To create a string of blanks, use **SPACE**.

Note that **LTRIM** can be used as an ODBC scalar function (with the curly brace syntax) or as an SQL general function.

Arguments

string-expression

A string expression, which can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR).

Examples

The following example removes the five leading blanks from the string. It leaves the five trailing blanks:

SQL

```
SELECT {fn LTRIM("     Test string with 5 leading and 5 trailing spaces.     ")}
```

Returns:

```
Before LTRIM
start:     Test string with 5 leading and 5 trailing spaces.     :end
After LTRIM
start:Test string with 5 leading and 5 trailing spaces.     :end
```

See Also

RTRIM TRIM LPAD SPACE

%MINUS (SQL)

A collation function that converts numbers to canonical collation format, then inverts the sign.

Synopsis

```
%MINUS(expression)

%MINUS expression
```

Description

%MINUS converts numbers or numeric strings to canonical form, inverts the sign, then returns these *expression* values in numeric collation sequence.

%MINUS and %PLUS are functionally identical, except that **%MINUS** inverts the sign. It prefixes a minus sign to any number that resolves to a positive number, and removes the minus sign from any number that resolves to a negative number. Zero is never signed.

A number can contain leading and trailing zeros, multiple leading plus and minus signs, a single decimal point indicator (.), and the E exponent indicator. In canonical form, all arithmetic operations are performed, exponents are expanded, signs are resolved to either a single leading minus sign or no sign, and leading and trailing zeros are stripped.

A numeric literal can be specified with or without enclosing string delimiters. If a string contains non-numeric characters, **%MINUS** truncates the number at the first non-numeric character, and returns the numeric part in canonical form. A non-numeric string (any string that begins with a non-numeric character) is returned as 0. **%MINUS** also returns NULLs as 0.

%MINUS is an InterSystems SQL extension and is intended for SQL lookup queries.

You can perform the same collation conversion in ObjectScript using the **Collation()** method of the %SYSTEM.Util class:

ObjectScript

```
WRITE $SYSTEM.Util.Collation("++007.500",4)
```

Compare **%MINUS** to **%MVR** collation, which sorts a string based on the numeric substrings within the string.

Arguments

expression

An expression, which can be the name of a column, a number or a string literal, an arithmetic expression, or the result of another function, where the underlying data type can be represented as any character type.

Example

The following example uses **%MINUS** to return records in descending numeric order of the home street number:

SQL

```
SELECT Name,Home_Street
FROM Sample.Person
ORDER BY %MINUS(Home_Street)
```

Note that the above example orders the integer part of the street address in numerical order. Compare this with the following ORDER BY DESC example, which orders records by street addresses in collation sequence:

SQL

```
SELECT Name,Home_Street
FROM Sample.Person
ORDER BY Home_Street DESC
```

See Also

- %EXACT collation function

- %PLUS collation function

- Collation

MINUTE (SQL)

A time function that returns the minute for a datetime expression.

Synopsis

```
{fn MINUTE(time-expression)}
```

Description

MINUTE returns an integer specifying the minutes for a given time or datetime value. Minutes are calculated for a $HOROLOG or $ZTIMESTAMP value, an ODBC format date string, or a timestamp.

A *time-expression* timestamp can be either data type %Library.PosixTime (an encoded 64-bit signed integer), or data type %Library.TimeStamp (yyyy-mm-dd hh:mm:ss.fff).

To change the default time format, use the SET OPTION command.

Note that you can supply a time integer (number of elapsed seconds), but not a time string (hh:mm:ss). You must supply a datetime string (yyyy-mm-dd hh:mm:ss).

The time portion of the datetime string must be a valid time. Otherwise, an SQLCODE -400 error <ILLEGAL VALUE> is generated. The minutes (mm) portion must be an integer in the range from 0 through 59. Leading zeros are optional on input; leading zeros are suppressed on output. You can omit the seconds (:ss) portion of a datetime string and still return the minutes portion.

The date portion of the datetime string is not validated.

MINUTE returns zero minutes when the minutes portion is '0' or '00'. Zero minutes is also returned if no time expression is supplied, or the minutes portion of the time expression is omitted entirely ('hh', 'hh:', 'hh::', or 'hh::ss').

The same time information can be returned using DATEPART or DATENAME.

This function can also be invoked from ObjectScript using the **MINUTE()** method call:

```
$SYSTEM.SQL.Functions.MINUTE(time-expression)
```

Arguments

time-expression

An expression that is the name of a column, the result of another scalar function, or a string or numeric literal. It must resolve either to a datetime string or a time integer, where the underlying data type can be represented as %Time, %TimeStamp, or %PosixTime.

Examples

The following examples both return the number 45 because it is the forty-fifth minute of the time expression in the datetime string:

SQL

```
SELECT {fn MINUTE('2018-02-16 18:45:38')} AS ODBCMinutes
```

SQL

```
SELECT {fn MINUTE(67538)} AS HorologMinutes
```

The following example also returns 45. As shown here, the seconds portion of the time value can be omitted:

SQL

```
SELECT {fn MINUTE('2018-02-16 18:45')} AS Minutes_Given
```

The following example returns 0 minutes because the time expression has been omitted from the datetime string:

SQL

```
SELECT {fn MINUTE('2018-02-16')} AS Minutes_Given
```

The following examples all return the minutes portion of the current time:

SQL

```
SELECT {fn MINUTE(CURRENT_TIME)} AS Min_CurrentT,
       {fn MINUTE({fn CURTIME()})} AS Min_CurT,
       {fn MINUTE({fn NOW()})} AS Min_Now,
       {fn MINUTE($HOROLOG)} AS Min_Horolog,
       {fn MINUTE($ZTIMESTAMP)} AS Min_ZTS
```

The following example shows that leading zeros are suppressed. The first **MINUTE** function returns a length 2, the others return a length of 1. An omitted time is considered to be 0 minutes, which has a length of 1:

SQL

```
SELECT LENGTH({fn MINUTE('2018-02-22 11:45:00')}),
       LENGTH({fn MINUTE('2018-02-22 03:05:00')}),
       LENGTH({fn MINUTE('2018-02-22 3:5:0')}),
       LENGTH({fn MINUTE('2018-02-22')})
```

The following Embedded SQL example shows that the **MINUTE** function recognizes the TimeSeparator character specified for the locale:

ObjectScript

```
DO ##class(%SYS.NLS.Format).SetFormatItem("TimeSeparator",".")
&sql(SELECT {fn MINUTE('2018-02-22 18.45.38')} INTO :a)
QUIT:(SQLCODE '= 0)
WRITE "minutes=",a
```

See Also

- SQL concepts: Data Type, Date and Time Constructs

- SQL functions: HOUR, SECOND, CURRENT_TIME, CURTIME, NOW, DATEPART, DATENAME

- ObjectScript function: $ZTIME

- ObjectScript special variables: $HOROLOG, $ZTIMESTAMP

MOD (SQL)

A scalar numeric function that returns the modulus (remainder) of a number divided by another.

Synopsis

```
MOD(dividend,divisor)
```

```
{fn MOD(dividend,divisor)}
```

Description

MOD returns the mathematical remainder (modulus) from the dividend by the divisor.

MOD can be specified as either a standard scalar function or an ODBC scalar function with curly brace syntax.

- If *dividend* and *divisor* are positive, it returns a positive modulo, or zero.

- If *dividend* and *divisor* are both negative, it returns a negative modulo, or zero.

- If *dividend* or *divisor* is NULL, it returns a NULL.

- If *divisor* is 0, it generates a SQLCODE -400 with a %msg <DIVIDE> error.

- If *divisor* is larger than *dividend*, it returns *dividend*.

The precision reported for **MOD** (either syntax form) is the same as the precision report for the arithmetic expression `dividend/divisor`.

ANSI Operator Precedence

The behavior of the **MOD** function with a single negative operand depends on the **Apply ANSI Operator Precedence** configuration setting:

- If **Apply ANSI Operator Precedence** is not applied, the behavior of **MOD** with a negative operand is the same as the # modulo operator. Both return the short count (the amount required to reach the next multiple), not the modulo. For example, 12#7 returns a modulo of 5; –12#7 returns a short count of 2. If *dividend* is negative, the short count is a positive value, or zero. If *divisor* is negative, the short count is a negative value, or zero.

- If **Apply ANSI Operator Precedence** is applied (the default at InterSystems IRIS 2019.1 and subsequent), the behavior of **MOD** with a negative operand is to always return a modulo. If *dividend* is negative, it returns a negative modulo, or zero. If *divisor* is negative, it returns a positive modulo, or zero.

The behavior of the # modulo operator is not affected by the **Apply ANSI Operator Precedence** configuration setting.

Arguments

dividend

A number that is the numerator (dividend) of the division.

divisor

A number that is the denominator (divisor) of the division.

MOD returns the NUMERIC data type unless the *dividend* is data type DOUBLE. If *dividend* is DOUBLE, **MOD** returns DOUBLE.

Examples

The following example shows the remainder returned by **MOD**.

SQL

```
SELECT MOD(5,3) AS Remainder
```

returns 2.

SQL

```
SELECT MOD(5.3,.5) AS Remainder
```

returns .3.

See Also

CEILING, FLOOR, ROUND, TRUNCATE

MONTH (SQL)

A date function that returns the month as an integer for a date expression.

Synopsis

```
MONTH(date-expression)
```

```
{fn MONTH(date-expression)}
```

Description

MONTH returns an integer specifying the month. The month integer is calculated for an InterSystems IRIS date integer, a $HOROLOG or $ZTIMESTAMP value, an ODBC format date string, or a timestamp.

A *date-expression* timestamp can be either data type %Library.PosixTime (an encoded 64-bit signed integer), or data type %Library.TimeStamp (yyyy-mm-dd hh:mm:ss.fff).

The month (mm) portion of a date string must be an integer in the range 1 through 12. Leading zeros are optional on input. Leading and trailing zeros are suppressed on output.

The date portion of *date-expression* is validated and must include a month within the range 1 through 12 and a valid day value for the specified month and year. Otherwise, an SQLCODE -400 error <ILLEGAL VALUE> is generated.

The time portion of *date-expression* is not validated and can be omitted.

Note that **MONTH** can be invoked as an ODBC scalar function (with the curly brace syntax) or as an SQL general function.

This function can also be invoked from ObjectScript using the **MONTH()** method call:

```
$SYSTEM.SQL.Functions.MONTH(date-expression)
```

The elements of a datetime string can be returned using the following SQL functions: **YEAR**, **MONTH**, **DAY** (or **DAYOFMONTH**), **HOUR**, **MINUTE**, and **SECOND**. The same elements can be returned by using the DATEPART or DATENAME function. Date elements can be returned using TO_DATE. **DATEPART** and **DATENAME** performs value and range checking on month values.

The LAST_DAY function returns the date of the last day of the specified month.

Arguments

date-expression

An expression that is the name of a column, the result of another scalar function, or a date or timestamp literal.

Examples

The following examples both return the number 2 because February is the second month of the year:

SQL

```
SELECT MONTH('2018-02-22') AS Month_Given
```

SQL

```
SELECT {fn MONTH(64701)} AS Month_Given
```

The following example sorts records in birthday order by month and day, ignoring the year component of the DOB:

SQL

```
SELECT Name,DOB AS Birthdays
FROM Sample.Person
ORDER BY MONTH(DOB),DAY(DOB),Name
```

The following examples all return the current month:

SQL

```
SELECT {fn MONTH({fn NOW()})} AS MNow,
       MONTH(CURRENT_DATE) AS MCurrD,
       {fn MONTH(CURRENT_TIMESTAMP)} AS MCurrTS,
       MONTH($HOROLOG) AS MHorolog,
       {fn MONTH($ZTIMESTAMP)} AS MZTS
```

See Also

- SQL functions: DATEPART, DATENAME, DAYOFMONTH, LAST_DAY, MONTHNAME, TO_DATE

- ObjectScript function: $ZDATE

- ObjectScript special variables: $HOROLOG, $ZTIMESTAMP

MONTHNAME (SQL)

A date function that returns the name of the month for a date expression.

Synopsis

```
{fn MONTHNAME(date-expression)}
```

Description

MONTHNAME takes as input an InterSystems IRIS date integer, a $HOROLOG or $ZTIMESTAMP value, an ODBC format date string, or a timestamp.

A *date-expression* timestamp can be either data type %Library.PosixTime (an encoded 64-bit signed integer), or data type %Library.TimeStamp (yyyy-mm-dd hh:mm:ss.fff).

The time portion of the timestamp is not evaluated and can be omitted.

MONTHNAME returns the name of the corresponding calendar month, January through December. The returned value is a character string with a maximum length of 15.

MONTHNAME checks that the date supplied is a valid date. The year must be between 0001 and 9999 (inclusive), the month 01 through 12, and the day appropriate for that month (for example, 02/29 is only valid on leap years). If the date is not valid, **MONTHNAME** issues an SQLCODE -400 <ILLEGAL VALUE> error.

The names of months default to the full-length American English month names. To change these month name values, use the SET OPTION command with the MONTH_NAME option.

The same month name information can be returned by using the DATENAME function. You can use TO_DATE to retrieve a month name or a month name abbreviation with other date elements. To return an integer corresponding to the month, use MONTH DATEPART or TO_DATE.

This function can also be invoked from ObjectScript using the **MONTHNAME()** method call:

```
$SYSTEM.SQL.Functions.MONTHNAME(date-expression)
```

Arguments

date-expression

An expression that evaluates to either an InterSystems IRIS date integer, an ODBC date, or a timestamp. This expression can be the name of a column, the result of another scalar function, or a date or timestamp literal.

Examples

The following examples both return the character string "February" because it is the month of the date expression (February 22, 2018):

SQL

```
SELECT {fn MONTHNAME('2018-02-22')} AS NameOfMonth
```

SQL

```
SELECT {fn MONTHNAME(64701)} AS NameOfMonth
```

The following examples all return the current month:

SQL

```
SELECT {fn MONTHNAME({fn NOW()})} AS MnameNow,
       {fn MONTHNAME(CURRENT_DATE)} AS MNameCurrDate,
       {fn MONTHNAME(CURRENT_TIMESTAMP)} AS MNameCurrTS,
       {fn MONTHNAME($HOROLOG)} AS MNameHorolog,
       {fn MONTHNAME($ZTIMESTAMP)} AS MNameZTS
```

The following example shows how **MONTHNAME** responds to an invalid date (the year 2017 was not a leap year):

SQL

```
SELECT {fn MONTHNAME("2017-02-29")}
```

The SQLCODE -400 error code is issued with the %msg indicating <ILLEGAL VALUE>.

See Also

- SQL functions: DATEPART, DATENAME, DAYOFMONTH, MONTH, TO_DATE

- ObjectScript function: $ZDATE

- ObjectScript special variables: $HOROLOG, $ZTIMESTAMP

NOW (SQL)

A date/time function that returns the current local date and time.

Synopsis

```
NOW()

{fn NOW}
{fn NOW()}
```

Description

NOW takes no arguments. The argument parentheses are optional for the ODBC scalar syntax; they are mandatory for the SQL standard function syntax.

NOW returns the current local date and time for this timezone as a timestamp; it adjusts for local time variants, such as Daylight Saving Time.

NOW can return a timestamp in either %TimeStamp data type format (yyyy-mm-dd hh:mm:ss.ffff) or %PosixTime data type format (an encoded 64-bit signed integer). The following rules determine which timestamp format is returned:

1. If the current timestamp is being supplied to a field of data type %PosixTime, the current timestamp value is returned in POSIXTIME data type format. For example, WHERE PosixField=NOW() or INSERT INTO MyTable (PosixField) VALUES (NOW()).

2. If the current timestamp is being supplied to a field of data type %TimeStamp, the current timestamp value is returned in TIMESTAMP data type format (yyyy-mm-dd hh:mm:ss). Its ODBC type is TIMESTAMP, LENGTH is 16, and PRECISION is 19. Hours are represented in 24–hour format. Leading zeros are retained for all fields. For example, WHERE TSField=NOW() or INSERT INTO MyTable (TSField) VALUES (NOW()).

3. If the current timestamp is being supplied without context, the current timestamp value is returned in TIMESTAMP data type format. For example, SELECT NOW().

To change the default datetime string format, use the SET OPTION command with the various date and time options.

You can use the CAST or CONVERT function to change the data type of timestamps, dates, and times.

Fractional Seconds of Precision

By default, **NOW** does not return fractional seconds of precision. It does not support a precision argument. However, by changing the system-wide default time precision, you can cause all **NOW** functions system-wide to return this configured number of digits of fractional second precision. The initial configuration setting of the system-wide default time precision is 0 (no fractional seconds); the highest setting is 9.

GETDATE is functionally identical to **NOW**, except that **GETDATE** provides a *precision* argument that allows you to override the system-wide default time precision; if you omit the *precision* argument, **GETDATE** takes the configured system-wide default time precision.

CURRENT_TIMESTAMP has two syntax forms: Without argument parentheses, **CURRENT_TIMESTAMP** is functionally identical to **NOW**. With argument parentheses, **CURRENT_TIMESTAMP(precision)**, is functionally identical to **GETDATE**, except that the **CURRENT_TIMESTAMP()** *precision* argument is mandatory. **CURRENT_TIMESTAMP()** always returns its specified *precision* and ignores the configured system-wide default time precision.

Fractional seconds are always truncated, not rounded, to the specified precision.

SYSDATE is functionally identical to the argumentless CURRENT_TIMESTAMP function.

Other Current Time and Date Functions

NOW, GETDATE, CURRENT_TIMESTAMP, and SYSDATE all return the current local date and time, based on the local time zone setting.

GETUTCDATE returns the current Universal Time Constant (UTC) date and time as a timestamp. Because UTC time does not depend on the local timezone and is not subject to local time variants (such as Daylight Saving Time), this function is useful for applying consistent timestamps when users in different time zones access the same database. **GETUTCDATE** supports fractional seconds of precision. The current UTC timestamp is also provided by the ObjectScript $ZTIMESTAMP special variable.

To return just the current date, use CURDATE or CURRENT_DATE. To return just the current time, use CURRENT_TIME or CURTIME. The functions use the DATE or TIME data type. The TIME and DATE data types store their values as integers in $HOROLOG format. None of these functions support precision.

Examples

The following example shows the three syntax forms are equivalent; all return the current local date and time as a timestamp:

SQL

```
SELECT NOW(),{fn NOW},{fn NOW()}
```

The following example compares local (time zone specific) and universal (time zone independent) timestamps:

SQL

```
SELECT NOW(), GETUTCDATE()
```

The following example sets the LastUpdate field in the selected row of the Orders table to the current system date and time:

SQL

```
UPDATE Orders SET LastUpdate = {fn NOW()}
  WHERE Orders.OrderNumber=:ord
```

See Also

- SQL concepts: Data Type, Date and Time Constructs

- SQL timestamp functions: CAST, CONVERT, CURRENT_TIMESTAMP, GETDATE, GETUTCDATE, SYSDATE, TIMESTAMPADD, TIMESTAMPDIFF, TO_TIMESTAMP

- SQL current date and time functions: CURDATE, CURRENT_DATE, CURRENT_TIME, CURTIME

- ObjectScript: $ZDATETIME function, $HOROLOG special variable, $ZTIMESTAMP special variable

NULLIF (SQL)

A function that returns NULL if two expressions have the same value.

Synopsis

```
NULLIF(expression1,expression2)
```

Description

The **NULLIF** function returns NULL if the value of *expression1* is equal to the value of *expression2*. Otherwise, it returns the *expression1* value.

NULLIF is equivalent to:

SQL

```
SELECT CASE
WHEN value1 = value2 THEN NULL
ELSE value1
END
FROM MyTable
```

Arguments

expression1

An expression, which can be the name of a column, a numeric or string literal, a host variable, or the result of another scalar function.

expression2

An expression, which can be the name of a column, a numeric or string literal, a host variable, or the result of another scalar function.

NULLIF returns the same data type as *expression1*.

NULL Handling Functions Compared

The following table shows the various SQL comparison functions. Each function returns one value if the logical comparison tests True (A same as B) and another value if the logical comparison tests False (A not same as B). These functions allow you to perform NULL logical comparisons. You cannot specify NULL in an actual equality (or non-equality) condition comparison.

SQL Function	Comparison Test	Return Value
NULLIF(ex1,ex2)	ex1 = ex2	True returns NULL False returns ex1
ISNULL(ex1,ex2)	ex1 = NULL	True returns ex2 False returns ex1
IFNULL(ex1,ex2) [two-argument form]	ex1 = NULL	True returns ex2 False returns NULL
IFNULL(ex1,ex2,ex3) [three-argument form]	ex1 = NULL	True returns ex2 False returns ex3
{fn IFNULL(ex1,ex2)}	ex1 = NULL	True returns ex2 False returns ex1
NVL(ex1,ex2)	ex1 = NULL	True returns ex2 False returns ex1
COALESCE(ex1,ex2,...)	ex = NULL for each argument	True tests next ex argument. If all ex arguments are True (NULL), returns NULL. False returns ex

Examples

The following example uses the **NULLIF** function to set to null the display field of all records with Age=20:

SQL

```
SELECT Name,Age,NULLIF(Age,20) AS Nulled20
FROM Sample.Person
```

See Also

- CASE command
- COALESCE function
- IFNULL function
- ISNULL function
- NVL function

NVL (SQL)

A function that tests for NULL and returns the appropriate expression.

Synopsis

```
NVL(check-expression,replace-expression)
```

Description

NVL evaluates *check-expression* and returns one of two values:

- If *check-expression* is NULL, *replace-expression* is returned.

- If *check-expression* is not NULL, *check-expression* is returned.

The arguments *check-expression* and *replace-expression* can have any data type. If their data types are different, SQL converts *replace-expression* to the data type of *check-expression* before comparing them. The data type of the return value is always the same as the data type of *check-expression*, unless *check-expression* is character data, in which case the return value's data type is VARCHAR2.

Note that **NVL** is supported for Oracle compatibility, and is the same as the **ISNULL** function.

Refer to NULL for further details on NULL handling.

DATE and TIME Display Conversion

Some *check-expression* data types require conversion from Logical mode to ODBC mode or Display mode. For example the DATE and TIME data types. If the *replace-expression* value is not the same data type, this value cannot be converted in ODBC mode or Display mode, and an SQLCODE error is generated: -146 for DATE data type; -147 for TIME data type. For example, `ISNULL(DOB,'nodate')` cannot be executed in ODBC mode or Display mode; it issue an SQLCODE -146 error with the %msg `Error: 'nodate' is an invalid ODBC/JDBC Date value` or `Error: 'nodate' is an invalid DISPLAY Date value`. To execute this statement in ODBC mode or Display mode, you must CAST the value as the appropriate data type: `ISNULL(DOB,CAST('nodate' as DATE))`. This results in a date 0, which displays as 1840-12-31.

Arguments

check-expression

The expression to be evaluated.

replace-expression

The expression that is returned if *check-expression* is NULL.

NVL returns the same data type as *check-expression*.

NULL Handling Functions Compared

The following table shows the various SQL comparison functions. Each function returns one value if the logical comparison tests True (A same as B) and another value if the logical comparison tests False (A not same as B). These functions allow you to perform NULL logical comparisons. You cannot specify NULL in an actual equality (or non-equality) condition comparison.

SQL Function	Comparison Test	Return Value
NVL(ex1,ex2)	ex1 = NULL	True returns ex2 False returns ex1
IFNULL(ex1,ex2) [two-argument form]	ex1 = NULL	True returns ex2 False returns NULL
IFNULL(ex1,ex2,ex3) [three-argument form]	ex1 = NULL	True returns ex2 False returns ex3
{fn IFNULL(ex1,ex2)}	ex1 = NULL	True returns ex2 False returns ex1
ISNULL(ex1,ex2)	ex1 = NULL	True returns ex2 False returns ex1
NULLIF(ex1,ex2)	ex1 = ex2	True returns NULL False returns ex1
COALESCE(ex1,ex2,...)	ex = NULL for each argument	True tests next ex argument. If all ex arguments are True (NULL), returns NULL. False returns ex

Examples

This following example returns the *replace-expression* (99) because the *check-expression* is NULL:

SQL

```
SELECT NVL(NULL,99) AS NullTest
```

This following example returns the *check-expression* (33) because *check-expression* is not NULL:

SQL

```
SELECT NVL(33,99) AS NullTest
```

The following example returns the string 'No Preference' if FavoriteColors is NULL; otherwise, it returns the value of FavoriteColors:

SQL

```
SELECT Name, NVL(FavoriteColors,'No Preference') AS ColorChoice
FROM Sample.Person
```

See Also

• CASE command

• COALESCE function

- [IFNULL](#) function
- [ISNULL](#) function
- [NULLIF](#) function

%OBJECT (SQL)

A scalar function that opens a stream object and returns the corresponding OREF.

Synopsis

```
%OBJECT(stream)
```

Description

%OBJECT is used to open a stream object and return the OREF (object reference) of the stream field.

A **SELECT** on a stream field returns the fully formed OID (object ID) value of the stream field. A **SELECT %OBJECT** on a stream field returns the OREF (object reference) of the stream field.

If *stream* is not a stream field, **%OBJECT** generates an SQLCODE -128 error.

%OBJECT can be used as an argument to the following functions:

- `CHARACTER_LENGTH(%OBJECT(streamfield))`, `CHAR_LENGTH(%OBJECT(streamfield))`, or `DATALENGTH(%OBJECT(streamfield))`.

- `SUBSTRING(%OBJECT(streamfield),start,length)`.

You can perform the same operation by issuing a **SELECT** on a stream field, then opening the stream OID by calling the **$Stream.Object.%Open()** class method, which generates an OREF from the OID:

```
SET oref = ##class(%Stream.Object).%Open(oid)
```

For information on OREFs, see OREF Basics. For information on OIDs, see Identifiers for Saved Objects: ID and OID.

Arguments

stream

An expression that is the name of a stream field.

Examples

The following example shows how a **SELECT %OBJECT** on a stream field returns the OREF, in which Notes and Picture are both stream fields:

SQL

```
SELECT TOP 3 Title,Notes,%OBJECT(Picture) AS Photo FROM Sample.Employee
```

See Also

- SELECT

- Introduction to the Default SQL Projection

- Using Streams with SQL

- Storing and Using BLOBs and CLOBs

%ODBCIN (SQL)

A format-transformation function that returns an expression in Logical format.

Synopsis

```
%ODBCIN(expression)

%ODBCIN expression
```

Description

%ODBCIN returns *expression* in the Logical format after passing the value through the field or data type's OdbcToLogical method. The Logical format is the in-memory format of data (the format upon which operations are performed).

%ODBCIN is an InterSystems SQL extension.

For further details on display format options, refer to Data Display Options.

Arguments

expression

The expression to be converted.

Examples

The following example shows the default display format, the **%ODBCIN**, and the **%ODBCOUT** formats for the same field.

SQL

```
SELECT FavoriteColors,%ODBCIN(FavoriteColors) AS InVal,
%ODBCOUT(FavoriteColors) AS OutVal
FROM Sample.Person
```

The following example uses **%ODBCIN** in the WHERE clause:

SQL

```
SELECT Name,DOB,%ODBCOUT(DOB) AS Birthdate
FROM Sample.Person
WHERE DOB BETWEEN %ODBCIN('2000-01-01') AND %ODBCIN('2018-01-01')
```

See Also

%EXTERNAL, %INTERNAL, %ODBCOUT

%ODBCOUT (SQL)

A format-transformation function that returns an expression in ODBC format.

Synopsis

```
%ODBCOUT(expression)

%ODBCOUT expression
```

Description

%ODBCOUT returns *expression* in the ODBC format after passing the value through the field or data type's LogicalToOdbc method. The ODBC format is the format in which data can be presented via ODBC. This format is used when data is exposed to ODBC/SQL. The available formats correspond to those defined by ODBC.

%ODBCOUT is commonly used on a **SELECT** list *select-item*. It can be used in a **WHERE** clause, but this use is discouraged because using **%ODBCOUT** prevents the use of indexes on the specified field.

Applying **%ODBCOUT** changes the column header name to a value such as "Expression_1"; it is therefore usually desirable to specify a column name alias, as shown in the examples below.

Whether **%ODBCOUT** converts a date depends on the data type returned by the date field or function. **%ODBCOUT** converts CURDATE, CURRENT_DATE, CURTIME, and CURRENT_TIME values. It does not convert CURRENT_TIMESTAMP, GETDATE, GETUTCDATE, NOW, and **$HOROLOG** values.

%ODBCOUT is an InterSystems SQL extension.

For further details on display format options, refer to Data Display Options.

Arguments

expression

The expression to be converted. A field name, an expression containing a field name, or a function that returns a value in a convertible data type, such as DATE or %List. Cannot be a stream field.

Examples

The following example shows the default display format, the **%ODBCIN**, and the **%ODBCOUT** formats for the same field.

SQL

```
SELECT FavoriteColors,%ODBCIN(FavoriteColors) AS InVal,
%ODBCOUT(FavoriteColors) AS OutVal
FROM Sample.Person
```

See Also

- %EXTERNAL, %INTERNAL, %ODBCIN

- SQL concepts: Data Types, Date and Time Constructs

%OID (SQL)

A scalar function that returns OID of an ID field.

Synopsis

```
%OID(id_field)
```

Description

%OID takes a field name and returns the fully formed OID (object ID) for the object. The field must be either an ID field or a reference field (a foreign key field). Specifying any other type of field in *id_field* generates an SQLCODE -1 error.

Arguments

id_field

The field name of an ID field, or a reference field.

Examples

The following example shows %OID used with a reference field:

SQL

```
SELECT Name, Spouse, %OID(Spouse)
FROM Sample.Person
WHERE Spouse IS NOT NULL
```

See Also

- SELECT

- %OBJECT

PI (SQL)

A scalar numeric function that returns the constant value of pi.

Synopsis

```
{fn PI()}
{fn PI}
```

Description

PI takes no arguments. It returns the mathematical constant pi as data type NUMERIC with a precision of 19 and a scale of 18.

PI can only be invoked using ODBC scalar function (curly brace) syntax. Note that the argument parentheses are optional.

Examples

The following examples both return the value of pi:

SQL

```
SELECT {fn PI()} AS ExactPi
```

SQL

```
SELECT {fn PI} AS ExactPi
```

returns 3.141592653589793238.

See Also

- SQL functions: ROUND

- ObjectScript special variable: $ZPI

$PIECE (SQL)

A string function that returns a substring identified by a delimiter.

Synopsis

```
$PIECE(string-expression,delimiter[,from[,to]])
```

Description

$PIECE returns the specified substring (piece) from *string-expression*. The substring returned depends on the arguments used:

- **$PIECE**(*string-expression*,*delimiter*) returns the first substring in *string-expression*. If *delimiter* occurs in *string-expression*, the substring that precedes the first occurrence of *delimiter* is returned. If *delimiter* does not occur in *string-expression*, the returned substring is *string-expression*.

- **$PIECE**(*string-expression*,*delimiter*,*from*) returns the substring which is the *n*th piece of *string-expression*, where the integer *n* is specified by the *from* argument, and pieces are separated by a *delimiter*. The delimiter is not returned.

- **$PIECE**(*string-expression*,*delimiter*,*from*,*to*) returns a range of substrings including the substring specified in *from* through the substring specified in *to*. This four-argument form of **$PIECE** returns a string that includes any intermediate occurrences of *delimiter* that occur between the *from* and *to* substrings. If *to* is greater than the number of substrings, the returned substring includes all substrings to the end of the *string-expression* string.

Arguments

string-expression

The string from which the substring is to be returned. It can be a string literal, a variable name, or any valid expression that evaluates to a string.

A string usually contains instances of a character (or character string) which are used as delimiters. This character or string cannot also be used as a data value within *string-expression*.

If you specify the null string (NULL) as the target string, **$PIECE** returns <null>, the null string.

delimiter

The search string to be used to delimit substrings within *string-expression*. It can be a numeric or string literal (enclosed in quotation marks), the name of a variable, or an expression that evaluates to a string.

Commonly, a delimiter is a designated character which is never used within string data, but is set aside solely for use as a delimiter separating substrings. A delimiter can also be a multi-character search string, the individual characters of which can be used within string data.

If you specify the null string (NULL) as the delimiter, **$PIECE** returns <null>, the null string.

from

An optional argument specifying the number of a substring within *string-expression*, counting from 1. It must be a positive integer, the name of an integer variable, or an expression that evaluates to a positive integer. Substrings are separated by delimiters.

- If the *from* argument is omitted or set to 1, **$PIECE** returns the first substring of *string-expression*. If *string-expression* does not contain the specified delimiter, a *from* value of 1 returns *string-expression*.

- If the *from* argument identifies by count the last substring in *string-expression*, this substring is returned, regardless of whether it is followed by a delimiter.

- If no *to* argument is specified and the value of *from* is NULL, the empty string, zero, or a negative number, **$PIECE** returns a null string. However, if a *to* argument is specified, **$PIECE** treats these *from* values the same as *from*=1.

- If the value of *from* is greater than the number of substrings in *string-expression*, **$PIECE** returns a null string.

If the *from* argument is used with the *to* argument, it identifies the start of a range of substrings to be returned as a string, and should be less than the value of *to*.

to

An optional argument specifying the number of the substring within *string-expression* that ends the range initiated by the *from* argument. The returned string includes both the *from* and *to* substrings, as well as any intermediate substrings and the delimiters separating them. The *to* argument must be a positive integer, the name of an integer variable, or an expression that evaluates to a positive integer. The *to* argument must be used with *from* and should be greater than the value of *from*.

- If *from* is less than *to*, **$PIECE** returns a string consisting of all of the delimited substrings within this range, including the *from* and *to* substrings. This returned string contains the substrings and the delimiters within this range.

- If *to* is greater than the number of delimited substrings, the returned string contains all the string data (substrings and delimiters) beginning with the *from* substring and continuing to the end of the *string-expression* string.

- If *from* is equal to *to*, the *from* substring is returned.

- If *from* is greater than *to*, **$PIECE** returns a null string.

- If *to* is the null string (NULL), **$PIECE** returns a null string.

Examples

The following example returns 'Red', the first substring as identified by the "," delimiter:

SQL

```
SELECT $PIECE('Red,Green,Blue,Yellow,Orange,Black',',')
```

The following example returns 'Blue', the third substring as identified by the "," delimiters:

SQL

```
SELECT $PIECE('Red,Green,Blue,Yellow,Orange,Black',',',3)
```

The following example returns 'Blue,Yellow,Orange', the third through fifth elements in *colorlist*, as delimited by ",":

SQL

```
SELECT $PIECE('Red,Green,Blue,Yellow,Orange,Black',',',3,5)
```

The following **$PIECE** functions both return '123', showing that the two-argument form is equivalent to the three-argument form when *from* is 1:

SQL

```
SELECT $PIECE('123#456#789','#') AS TwoArg
```

SQL

```
SELECT $PIECE('123#456#789','#',1) AS ThreeArg
```

The following example uses the multi-character delimiter string '#-#' to return the third substring '789'. Here, the component characters of the delimiter string, '#' and '-', can be used as data values; only the specified sequence of characters (#-#) is set aside:

SQL

```
SELECT $PIECE('1#2-3#-#45##6#-#789','#-#',3)
```

The following example returns 'MAR;APR;MAY'. These comprise the third through the fifth substrings, as identified by the ';' *delimiter*:

SQL

```
SELECT $PIECE('JAN;FEB;MAR;APR;MAY;JUN',';',3,5)
```

The following example uses **$PIECE** to extract the surname from employee names and vendor contact names, and then perform a JOIN which return instances where an employee has the same surname as a vendor contact:

SQL

```
SELECT E.Name,V.Contact
FROM Sample.Employee AS E INNER JOIN Sample.Vendor AS V
ON $PIECE(E.Name,',')=$PIECE(V.Contact,',')
```

Notes

Using $PIECE to Unpack Data Values

$PIECE is typically used to "unpack" data values that contain multiple fields delimited by a separator character. Typical delimiter characters include the slash (/), the comma (,), the space ( ), and the semicolon (;). The following sample values are good candidates for use with **$PIECE**:

```
'John Jones/29 River St./Boston MA, 02095'
'Mumps;Measles;Chicken Pox;Diptheria'
'45.23,52.76,89.05,48.27'
```

$PIECE and $LENGTH

The two-argument form of **$LENGTH** returns the number of substrings in a string, based on a delimiter. Use **$LENGTH** to determine the number of substrings in a string, and then use **$PIECE** to extract individual substrings.

$PIECE and $LIST

The data storage techniques used by **$PIECE** and the **$LIST** functions are incompatible and should not be combined. For example, attempting to use **$PIECE** on a list created using **$LISTBUILD** yields unpredictable results and should be avoided. This is true for both SQL functions and the corresponding ObjectScript functions.

The **$LIST** functions specify substrings without using a designated delimiter. If setting aside a delimiter character or character sequence is not appropriate to the type of data (for example, bitstring data), you should use the **$LISTBUILD** and **$LIST** SQL functions to store and retrieve substrings.

Null Values

$PIECE does not distinguish between a delimited substring with a null string value (NULL), and a nonexistent substring. Both return <null>, the null string value. For example, the following examples both return the null string for a *from* value of 7:

SQL

```
SELECT $PIECE('Red,Green,Blue,Yellow,Orange,Black',',',7)
```

SQL

```
SELECT $PIECE('Red,Green,Blue,Yellow,Orange,Black,',',',7)
```

In the first case, there is no seventh substring; a null string is returned. In the second case there is a seventh substring, as indicted by the delimiter at the end of the *string-expression* string; the value of this seventh substring is the null string.

The following example shows null values within a *string-expression*. It extracts substrings 3. This substring exists, but contains a null string:

SQL

```
SELECT $PIECE('Red,Green,,Blue,Yellow,Orange,Black,',',',3)
```

The following examples also returns a null string, because the specified substrings do not exist:

SQL

```
SELECT $PIECE('Red,Green,,Blue,Yellow,Orange,Black,',',',0)
```

SQL

```
SELECT $PIECE('Red,Green,,Blue,Yellow,Orange,Black,',',',8,20)
```

In the following example, the **$PIECE** function returns the entire *string-expression* string, because there are no occurrences of *delimiter* in the *string-expression* string:

SQL

```
SELECT $PIECE('Red,Green,Blue,Yellow,Orange,Black,','#')
```

Nested $PIECE Operations

To perform complex extractions, you can nest **$PIECE** references within each other. The inner **$PIECE** returns a substring that is operated on by the outer **$PIECE**. Each **$PIECE** uses its own delimiter. For example, the following returns the state abbreviation 'MA':

SQL

```
SELECT $PIECE($PIECE('John Jones/29 River St./Boston MA 02095','/',3),' ',2)
```

The following is another example of nested **$PIECE** operations, using a hierarchy of delimiters. First, the inner **$PIECE** uses the caret (^) delimiter to find the second piece, 'A,B,C', of the string. Then the outer **$PIECE** uses the comma (,) delimiter to return the first and second pieces ('A,B') of the substring 'A,B,C':

SQL

```
SELECT $PIECE($PIECE('1,2,3^A,B,C^@#!','^',2),',',1,2)
```

See Also

- SQL functions: $EXTRACT $FIND $LENGTH $LIST

- ObjectScript functions: $EXTRACT $FIND $LENGTH $LIST $PIECE

%PLUS (SQL)

A collation function that converts numbers to canonical collation format.

Synopsis

```
%PLUS(expression)

%PLUS expression
```

Description

%PLUS converts numbers or numeric strings to canonical form, then returns these *expression* values in numeric collation sequence.

A number can contain leading and trailing zeros, multiple leading plus and minus signs, a single decimal point indicator (.), and the E exponent indicator. In canonical form, all arithmetic operations are performed, exponents are expanded, signs are resolved to either a single leading minus sign or no sign, and leading and trailing zeros are stripped.

A numeric literal can be specified with or without enclosing string delimiters. If a string contains non-numeric characters, **%PLUS** truncates the number at the first non-numeric character, and returns the numeric part in canonical form. A non-numeric string (any string that begins with a non-numeric character) is returned as 0. **%PLUS** also returns NULLs as 0.

%PLUS is an InterSystems SQL extension and is intended for SQL lookup queries.

You can perform the same collation conversion in ObjectScript using the **Collation**() method of the %SYSTEM.Util class:

ObjectScript

```
WRITE $SYSTEM.Util.Collation("++007.500",3)
```

Compare **%PLUS** to **%MVR** collation, which sorts a string based on the numeric substrings within the string.

Arguments

expression

An expression, which can be the name of a column, a number or a string literal, an arithmetic expression, or the result of another function, where the underlying data type can be represented as any character type.

Examples

The following examples uses **%PLUS** to return Home_Street addresses in numeric order:

SQL

```
SELECT Name,Home_Street
FROM Sample.Person
ORDER BY %PLUS(Home_Street)
```

Note that the above example orders the integer part of the street address in ascending numerical order. Compare this with the following **ORDER BY** example, which orders records by street addresses in collation sequence:

SQL

```
SELECT Name,Home_Street
FROM Sample.Person
ORDER BY Home_Street
```

See Also

- %EXACT collation function

- %MINUS collation function

- Collation

POSITION (SQL)

A string function that returns the position of a substring within a string.

Synopsis

```
POSITION(substring IN string)
```

Description

POSITION returns the position of the first location of *substring* within *string*. The position is returned as an integer. If *substring* is not found, 0 (zero) is returned. If a NULL value is passed for either argument, **POSITION** returns NULL.

POSITION is case-sensitive. Use one of the case-conversion functions to locate both uppercase and lowercase instances of a letter or character string.

POSITION, INSTR, CHARINDEX, and $FIND

POSITION, **INSTR**, **CHARINDEX**, and **$FIND** all search a string for a specified substring and return an integer position corresponding to the first match. **CHARINDEX**, **POSITION**, and **INSTR** return the integer position of the first character of the matching substring. **$FIND** returns the integer position of the first character after the end of the matching substring. **CHARINDEX**, **$FIND**, and **INSTR** support specifying a starting point for substring search. **INSTR** also supports specifying the substring occurrence from that starting point.

The following example demonstrates these four functions, specifying all optional arguments. Note that the positions of *string* and *substring* differ in these functions:

SQL

```
SELECT POSITION('br' IN 'The broken brown briefcase') AS Position,
       CHARINDEX('br','The broken brown briefcase',6) AS Charindex,
       $FIND('The broken brown briefcase','br',6) AS Find,
       INSTR('The broken brown briefcase','br',6,2) AS Inst
```

For a list of functions that search for a substring, refer to String Manipulation.

Arguments

substring

The substring to search for. It can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR2).

IN string

The string expression within which to search for *substring*.

Examples

The following example returns 11, because "b" is the 11th character in the string:

SQL

```
SELECT POSITION('b' IN 'The quick brown fox') AS PosInt
```

The following example returns the length of the last name (surname) for each name in the Sample.Person table. It locates the comma used to separate the last name from the rest of the name field, then subtracts 1 from that position:

SQL

```
SELECT Name,
POSITION(',' IN Name)-1 AS LNameLen
FROM Sample.Person
```

The following example returns the position of the first instance of the letter "B" in each name in the Sample.Person table. Because **POSITION** is case-sensitive, the **%SQLUPPER** function is used to convert all name values to uppercase before performing the search. Because **%SQLUPPER** adds a blank space at the beginning of a string, this example subtracts 1 to get the actual letter position. Searches that do not locate the specified string return zero (0); in this example, because of the subtraction of 1, the value displayed for these searches is –1:

SQL

```
SELECT Name,
POSITION('B' IN %SQLUPPER(Name))-1 AS BPos
FROM Sample.Person
```

See Also

- CHARINDEX function

- $FIND function

- INSTR function

- String Manipulation

POWER (SQL)

A numeric function that returns the value of a given expression raised to the specified power.

Synopsis

```
POWER(numeric-expression,power)

{fn POWER(numeric-expression,power)}
```

Description

POWER calculates one number raised to the power of another. It returns a value with a precision of 36 and a scale of 18.

Note that **POWER** can be invoked as an ODBC scalar function (with the curly brace syntax) or as an SQL general scalar function.

POWER interprets a non-numeric string as 0 for either argument. For further details, refer to Strings as Numbers. **POWER** returns NULL if passed a NULL value for either argument.

All combinations of *numeric-expression* and *power* are valid except:

- `POWER(0,-m)`: a 0 *numeric-expression* and a negative *power* results in an SQLCODE -400 error.

- `POWER(-n,.m)`: a negative *numeric-expression* and a fractional *power* results in an SQLCODE -400 error.

Arguments

numeric-expression

The base number. Can be a positive or negative integer or fractional number.

power

The exponent, which is the power to which to raise *numeric-expression*. Can be a positive or negative integer or fractional number.

POWER returns either the NUMERIC or DOUBLE data type. If *numeric-expression* is data type DOUBLE, **POWER** returns DOUBLE; otherwise, it returns NUMERIC.

Examples

The following example raises 5 to the 3rd power:

SQL

```
SELECT POWER(5,3) AS Cubed
```

returns 125.

The following embedded SQL example returns the first 16 powers of 2:

ObjectScript

```
SET a=1
WHILE a<17 {
&sql(SELECT {fn POWER(2,:a)}
INTO :b)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE
  QUIT }
ELSE {
  WRITE !,"2 to the ",a," = ",b
  SET a=a+1 }
}
```

See Also

- SQL functions: EXP LOG10 SQRT SQUARE

- ObjectScript function: $ZPOWER

- ObjectScript Exponentiation Operator (**)

PREDICT (SQL)

A function that applies a specified trained model to predict the result for each input row provided.

Synopsis

```
PREDICT( model-name)
PREDICT( model-name USE  trained-model-name )
PREDICT( model-name WITH feature-columns-clause )
PREDICT( model-name USE  trained-model-name WITH feature-columns-clause )
```

Description

PREDICT returns the result of applying a trained machine learning model onto a specified query. This is performed on a row-by-row basis.

USE

If a trained model is not explicitly named by USE, **PREDICT** uses the default trained model for the specified model definition.

For example, if multiple models are trained:

```
CREATE MODEL MyModel PREDICTING( label ) FROM data
TRAIN MODEL MyModel AS FirstModel
TRAIN MODEL MyModel AS SecondModel NOT DEFAULT
```

`FirstModel` is the default model for `MyModel`. This means that **PREDICT** queries would use `FirstModel` for predictions. To specify use of `SecondModel`:

```
PREDICT( MyModel USE SecondModel)
```

WITH

When using a **FROM** clause to specify a dataset, the **PREDICT** function implicitly maps the feature columns of the specified dataset to those in the model. You can use a **WITH** clause to either:

- Specify the mapping of columns between the dataset and your model. For example:

  ```
  SELECT PREDICT(Trained_Model WITH age = year) FROM dataset
  ```

  This query matches the `age` column from `Trained_Model` to the `year` column from `dataset`.

  You can use braces to map multiple columns:

  ```
  SELECT PREDICT(Trained_Model WITH {age = year, income = salary}) FROM dataset
  ```

  The order of these columns in your **WITH** clause does not matter, and any missing column names are taken from the **FROM** clause.

- Specify a list of arguments to make a prediction with. When using this form of **WITH**, you do not provide a **FROM** clause. For example:

  ```
  SELECT PREDICT(Flower_Model WITH (5.1, 3.5, 1.4, 0.2, 'setosa'))
  ```

  This query makes a prediction using `Flower_Model` on the expression `(5.1, 3.5, 1.4, 0.2, 'setosa')`.

  Arguments must be ordered exactly as specified in your **CREATE MODEL** statement. Missing arguments can be specified by empty commas. For example:

  ```
  SELECT PREDICT(Flower_Model WITH (5.1, , 1.4, , 'setosa'))
  ```

You can use braces to provide multiple sets of arguments:

```
SELECT PREDICT(Flower_Model WITH ({5.1, 3.5, 1.4, 0.2, 'setosa'}, {6.4, 3.2, 4.3, 1.2, 'versicolor'}))
```

Required Security Privileges

Calling **PREDICT** requires %USE_MODEL privileges; otherwise, there is a SQLCODE –99 error (Privilege Violation). To assign %USE_MODEL privileges, use the GRANT command.

Arguments

model-name

The name of the model.

USE trained-model-name

An optional argument that specifies the name of a non-default trained model. See details above.

WITH feature-columns-clause

An optional argument. The specific columns to provide as input for your trained model. See details above.

Examples

```
CREATE MODEL HousePriceModel PREDICTING( HousePrice ) FROM housing_data_2019
TRAIN MODEL HousePriceModel
SELECT * FROM housing_data_2020 WHERE PREDICT( HousePriceModel ) > 500000

CREATE MODEL PatientReadmission PREDICTING ( IsReadmitted ) FROM patient_data
TRAIN MODEL PatientReadmission
SELECT *, PREDICT( PatientReadmission ) FROM new_patient_data
```

See Also

*   TRAIN MODEL, PROBABILITY

PROBABILITY (SQL)

A function that applies a specified trained model to return the probability that the specified label is the predicted label value. This allows you to evaluate the relative strength of predictions of that value.

Synopsis

```
PROBABILITY ( model-name FOR label-value)
PROBABILITY ( model-name USE  trained-model-name FOR label-value )
PROBABILITY ( model-name FOR label-value
      WITH  feature-columns-clause )
PROBABILITY ( model-name USE  trained-model-name FOR label-value
      WITH feature-columns-clause ] )
```

Description

The **PROBABILITY** function applies a given model to a given table, returning the probability that, for each row in the table, the model would predict the specified value. This probability is returned as a value from 0 to 1. This function can only be used with classification models (not regression models).

FOR

FOR provides the output value that **PROBABILITY** finds the probability of.

For example:

```
SELECT * FROM flower_dataset WHERE PROBABILITY(iris_flower FOR 'iris-setosa') > 0.6
```

Uses the `iris_flower` model to return each row in `flower_dataset` where the probability of the result being "iris-setosa" is greater than 0.6.

Omitting **FOR** implies a value of 1. For example:

```
SELECT PROBABILITY(IsSpam) FROM email_data
```

Implicitly forms this query:

```
SELECT PROBABILITY(IsSpam FOR 1) FROM email_data
```

When the value provided for **FOR** is invalid for the specified trained model, there is a SQLCODE –400 error with the following message:

```
[%msg: <PREDICT execution error: ERROR #2853: Specified positive label value not found
in the dataset.>]
```

USE

If a trained model is not explicitly named by USE, **PROBABILITY** uses the default trained model for the specified model definition.

For example, if multiple models are trained:

```
CREATE MODEL MyModel PREDICTING( label ) FROM data
TRAIN MODEL MyModel AS FirstModel
TRAIN MODEL MyModel AS SecondModel NOT DEFAULT
```

`FirstModel` is the default model for `MyModel`. This means that **PROBABILITY** queries would use `FirstModel` for predictions. To specify use of `SecondModel`:

```
PROBABILITY( MyModel FOR label-value USE SecondModel)
```

WITH

PROBABILITY is a smart function, mapping the feature columns of the specified dataset to those in the model implicitly when there is no **WITH** clause. You can use a **WITH** clause to specify the mapping of columns between the dataset and your model. For example:

```
SELECT PROBABILITY(iris_flower FOR 'iris-setosa' WITH petal_length = length_petal) FROM flower_dataset
```

This query matches the `petal_length` column from the `iris_flower` model to the `length_petal` column from `flower_dataset`.

Required Security Privileges

Calling **PROBABILITY** requires %USE_MODEL privileges; otherwise, there is a SQLCODE –99 error (Privilege Violation). To assign %USE_MODEL privileges, use the GRANT command.

Arguments

model-name

The name of the trained model.

FOR label-value

The output value. See details above.

USE trained-model-name

An optional argument. The name of a non-default trained model. See details above.

WITH feature-columns-clause

An optional argument. The specific columns to provide as input for your trained model. See details above.

Examples

```
CREATE MODEL PatientReadmission PREDICTING ( IsReadmitted ) FROM patient_data
TRAIN MODEL PatientReadmission
SELECT * FROM new_patient_data WHERE PROBABILITY( PatientReadmission FOR 1) > 0.8
```

See Also

*   TRAIN MODEL, PREDICT

QUARTER (SQL)

A date function that returns the quarter of the year as an integer for a date expression.

Synopsis

```
{fn QUARTER(date-expression)}
```

Description

QUARTER returns an integer from 1 to 4. The quarter is calculated for an InterSystems IRIS date integer, a $HOROLOG or $ZTIMESTAMP value, an ODBC format date string, or a timestamp.

A *date-expression* timestamp can be either data type %Library.PosixTime (an encoded 64-bit signed integer), or data type %Library.TimeStamp (yyyy-mm-dd hh:mm:ss.fff).

The time periods for the four quarters are as follows:

Quarter	Period (inclusive)
1	January 1 to March 31 (90 or 91 days)
2	April 1 to June 30 (91 days)
3	July 1 to September 30 (92 days)
4	October 1 to December 31 (92 days)

QUARTER is based on the month portion of a datetime string. However, all of *date-expression* is validated and must include a month within the range 1 through 12 and a valid day value for the specified month and year. Otherwise, an SQLCODE -400 error <ILLEGAL VALUE> is generated. The time portion of *date-expression* can be omitted, but if present must be valid.

The same quarter information can be returned by using the DATEPART or DATENAME function. You can use the DATEADD or TIMESTAMPADD function to increment a date by a specified number of quarters.

This function can also be invoked from ObjectScript using the **QUARTER()** method call:

```
$SYSTEM.SQL.Functions.QUARTER(date-expression)
```

Arguments

date-expression

An expression that is the name of a column, the result of another scalar function, or a date or timestamp literal.

Examples

The following examples both return the number 1 because the date (February 22) is in the first quarter of the year:

SQL

```
SELECT {fn QUARTER('2018-02-22')} AS ODBCDateQ
```

SQL

```
SELECT {fn QUARTER(64701)} AS HorologDateQ
```

The following examples all return the current quarter:

SQL

```
SELECT {fn QUARTER({fn NOW()})} AS Q_Now,
       {fn QUARTER(CURRENT_DATE)} AS Q_CurrD,
       {fn QUARTER(CURRENT_TIMESTAMP)} AS Q_CurrTstamp,
       {fn QUARTER($ZTIMESTAMP)} AS Q_ZTstamp,
       {fn QUARTER($HOROLOG)} AS Q_Horolog
```

See Also

- SQL functions: DATENAME, DATEPART, DATEADD, MONTH, TO_DATE

- ObjectScript function: $ZDATE

- ObjectScript special variables: $HOROLOG, $ZTIMESTAMP

RADIANS (SQL)

A numeric function that converts degrees to radians.

Synopsis

```
RADIANS(numeric-expression)

{fn RADIANS(numeric-expression)}
```

Description

RADIANS takes an angle measurement in degrees and returns the corresponding angle measurement in radians. **RADIANS** returns NULL if passed a NULL value.

The returned value has a default precision of 36 and a default scale of 18.

You can use the DEGREES function to convert radians to degrees.

Arguments

numeric-expression

The measure of an angle in degrees. An expression that resolves to a numeric value.

RADIANS returns either the NUMERIC or DOUBLE data type. If *numeric-expression* is data type DOUBLE, **RADIANS** returns DOUBLE; otherwise, it returns NUMERIC.

RADIANS can be specified as either a standard scalar function or an ODBC scalar function with curly brace syntax.

Example

The following Embedded SQL example returns the radians equivalents corresponding to the degree values from 0 through 365 in 30-degree increments:

ObjectScript

```
SET a=0
WHILE a<366 {
&sql(SELECT RADIANS(:a) INTO :b)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE
  QUIT }
ELSE {
  WRITE !,"degrees ",a," = radians ",b
  SET a=a+30 }
}
```

See Also

- SQL functions: CONVERT, DEGREES, TO_NUMBER

REPEAT (SQL)

A string function that repeats a string a specified number of times.

Synopsis

```
REPEAT(expression,repeat-count)
```

```
{fn REPEAT(expression,repeat-count)}
```

Description

REPEAT returns a string of *repeat-count* instances of *expression*, concatenated together.

If *expression* is NULL, **REPEAT** returns NULL. If *expression* is the empty string, **REPEAT** returns an empty string.

If *repeat-count* is a fractional number, only the integer part is used. If *repeat-count* is 0, **REPEAT** returns an empty string. If *repeat-count* is a negative number, NULL, or a non-numeric string, **REPEAT** returns NULL.

Arguments

expression

The string expression to be repeated.

repeat-count

The number of times to repeat, expressed as an integer.

Examples

The following examples show the two forms of **REPEAT**. Both examples return the string 'BANGBANGBANG':

SQL

```
SELECT REPEAT('BANG',3) AS Tripled
```

SQL

```
SELECT {fn REPEAT('BANG',3)} AS Tripled
```

See Also

- REPLICATE

REPLACE (SQL)

A string function that replaces a substring within a string.

Synopsis

```
REPLACE(string,oldsubstring,newsubstring)
```

Description

REPLACE searches a string for a substring and replaces all matches. Matching is case-sensitive. If a match is found, it replaces every instance of *oldsubstring* with *newsubstring*. The replacement substring may be longer or shorter than the substring it replaces. If the substring cannot be found, **REPLACE** returns the original *string* unchanged.

The value returned by **REPLACE** is always of data type VARCHAR, regardless of the data type of *string*. This allows for replacement operations such as REPLACE(12.3,'.','_').

REPLACE cannot use a %Stream.GlobalCharacter field for the *string*, *oldsubstring*, or *newsubstring* argument. Attempting to do so generates an SQLCODE -37 error.

The empty string is a string value. You can, therefore, use the empty string for any argument value. However, note that the ObjectScript empty string is passed to InterSystems SQL as NULL.

NULL is not a data value in InterSystems SQL. For this reason, specifying NULL for any of the **REPLACE** arguments returns NULL, regardless of whether or not a match occurs.

This function provides compatibility with Transact-SQL implementations.

REPLACE, STUFF, and $TRANSLATE

Both **REPLACE** and **STUFF** perform substring replacement. **REPLACE** searches for a substring by data value. **STUFF** searches for a substring by string position and length.

REPLACE performs a single string-for-string matching and replacement. **$TRANSLATE** performs character-for-character matching and replacement; it can replace all instances of one or more specified single characters with corresponding specified replacement single characters. It can also remove all instances of one or more specified single characters from a string.

By default, all three functions are case-sensitive and replace all matching instances.

For a list of functions that search for a substring, refer to String Manipulation.

Arguments

string

A string expression that is the target for the substring search.

oldsubstring

The substring to match within *string*.

newsubstring

The substring used to replace *oldsubstring*.

Examples

The following example searches for every instance of the substring 'P' and replaces it with the substring 'K':

SQL

```
SELECT REPLACE('PING PONG','P','K')
```

The following example searches for every instance of the substring 'KANSAS' and replaces it with the substring 'NEBRASKA':

SQL

```
SELECT REPLACE('KANSAS, ARKANSAS, NEBRASKA','KANSAS','NEBRASKA')
```

The following example show that **REPLACE** handles the empty string (") just like any other string value:

SQL

```
SELECT REPLACE('','','Nothing'),
       REPLACE('PING PONG','','K'),
       REPLACE('PING PONG','P','')
```

The following example shows that **REPLACE** handles any NULL argument by returning NULL. All of the following **REPLACE** functions return NULL, including the last, in which no match occurs:

SQL

```
SELECT REPLACE(NULL,'K','P'),
       REPLACE(NULL,NULL,'P'),
       REPLACE('PING PONG',NULL,'K'),
       REPLACE('PING PONG','P',NULL),
       REPLACE('PING PONG','Z',NULL)
```

The following Embedded SQL example is identical to the previous NULLs example. It shows how the ObjectScript empty string host variable is treated as NULL within SQL:

ObjectScript

```
 SET a=""
 &sql(SELECT
 REPLACE(:a,'K','P'),
 REPLACE(:a,:a,'P'),
 REPLACE('PING PONG',:a,'K'),
 REPLACE('PING PONG','P',:a),
 REPLACE('PING PONG','Z',:a)
 INTO :v,:w,:x,:y,:z)
 WRITE !,"SQLCODE=",SQLCODE
 WRITE !,"Output string=",v
 WRITE !,"Output string=",w
 WRITE !,"Output string=",x
 WRITE !,"Output string=",y
 WRITE !,"Output string=",z
```

See Also

- [CHARINDEX](#) function

- [$FIND](#) function

- [STUFF](#) function

- [$TRANSLATE](#) function

- [String Manipulation](#)

REPLICATE (SQL)

A string function that repeats a string a specified number of times.

Synopsis

```
REPLICATE(expression,repeat-count)
```

Description

REPLICATE repeats a string a specified number of times.

Note: The **REPLICATE** function is an alias for the **REPEAT** function. **REPLICATE** is provided for TSQL compatibility. Refer to REPEAT for further details.

Arguments

expression

The string expression to be repeated.

repeat-count

The number of times to repeat, expressed as an integer.

See Also

- REPEAT

REVERSE (SQL)

A scalar string function that returns a character string in reverse character order.

Synopsis

```
REVERSE(string-expression)
```

Description

REVERSE returns *string-expression* with its character order reversed. For example, 'Hello World!' is returned as '!dlroW olleH'. This is a simple string-order reversal, with no additional processing.

The string returned is data type VARCHAR, regardless of the data type of the input value. Numbers are converted to canonical form, numeric strings are not converted to canonical form before reversing.

Leading and trailing blanks are unaffected by reversing.

Reversing a NULL value results in a NULL.

Note: Because **REVERSE** always returns a VARCHAR string, some types of data become invalid when reversed:

- A reversed list is no longer a valid list and cannot be converted from storage format to display format.

- A reversed date is no longer a valid date, and cannot be converted from storage format to display format.

Arguments

string-expression

The string expression to be reversed. The expression can be the name of a column, a string literal, a numeric, or the result of another scalar function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR).

Examples

The following example reverses the Name field values. In this case, this results in names sorted by middle initial:

SQL

```
SELECT Name,REVERSE(Name) AS RevName
FROM Sample.Person
ORDER BY RevName
```

Note that because Name and RevName are just different representations of the same field, `ORDER BY RevName` and `ORDER BY RevName,Name` perform the same ordering.

The following example reverses a number and a numeric string:

SQL

```
SELECT REVERSE(+007.10) AS RevNum,
       REVERSE('+007.10') AS RevNumStr
```

The following example reverses a $DOUBLE number:

SQL

```
SELECT
  CAST(1.1 AS DOUBLE) AS DoubleNumber,
  REVERSE(CAST(1.1 AS DOUBLE)) AS ReverseDouble
```

The following example shows what happens when you reverse a list:

SQL

```
SELECT FavoriteColors,REVERSE(FavoriteColors) AS RevColors
FROM Sample.Person
```

The following example shows what happens when you reverse a date:

SQL

```
SELECT DOB,%INTERNAL(DOB) AS IntDOB,REVERSE(DOB) AS RevDOB
FROM Sample.Person
```

See Also

- CHAR

- STRING

- SUBSTRING

RIGHT (SQL)

A scalar string function that returns a specified number of characters from the end (rightmost position) of a string expression.

Synopsis

```
{fn RIGHT(string-expression,count)}
```

Description

RIGHT returns *count* number of characters from the end (rightmost position) of *string-expression*. **RIGHT** returns NULL if passed a NULL value for either argument.

RIGHT can only be used as an ODBC scalar function (with the curly brace syntax).

Arguments

string-expression

A string expression, which can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR).

count

An integer that specifies the number of characters to return from the ending (rightmost) position of *string-expression*.

Examples

The following example returns the two rightmost characters of each name in the Sample.Person table:

SQL

```
SELECT Name,{fn RIGHT(Name,2)}AS MiddleInitial
    FROM Sample.Person
```

The following example shows how **RIGHT** handles a *count* that is longer than the string itself:

SQL

```
SELECT Name,{fn RIGHT(Name,40)} FROM Sample.Person
```

No padding is performed.

See Also

- LEFT LTRIM RTRIM

ROUND (SQL)

A numeric function that rounds or truncates a number at a specified number of digits.

Synopsis

```
ROUND(numeric-expr,scale[,flag])  {fn ROUND(numeric-expr,scale[,flag])}
```

Description

This function can be used to either round or truncate a number to the specified number of decimal digits.

ROUND rounds or truncates *numeric-expr* to *scale* places, counting from the decimal point. When rounding, the number 5 is always rounded up. Trailing zeroes are removed after a **ROUND** round or truncate operation. Leading zeros are not returned.

- If *scale* is a positive number, rounding is performed at that number of digits to the right of the decimal point. If *scale* is equal to or larger than the number of decimal digits, no rounding or zero filling occurs.

- If *scale* is zero, rounding is to the closest whole integer. In other words, rounding is performed at zero digits to the right of the decimal point; all decimal digits and the decimal point itself are removed.

- If *scale* is a negative number, rounding is performed at that number of digits to the left of the decimal point. If *scale* is equal to or larger than the number of integer digits in the rounded result, zero is returned.

- If *numeric-expr* is zero (however expressed: 00.00, -0, etc.) **ROUND** returns 0 (zero) with no decimal digits, regardless of the *scale* value.

- If *numeric-expr* or *scale* is NULL, **ROUND** returns NULL.

Note that the **ROUND** return value is always normalized, removing trailing zeros.

ROUND, TRUNCATE, and $JUSTIFY

ROUND and **TRUNCATE** are numeric functions that perform similar operations; they both can be used to decrease the number of significant decimal or integer digits of a number. **ROUND** allows you to specify either rounding (the default), or truncation; **TRUNCATE** does not perform rounding. **ROUND** returns the same data type as *numeric-expr*; **TRUNCATE** returns *numeric-expr* as data type NUMERIC, unless *numeric-expr* is data type DOUBLE, in which case it returns data type DOUBLE.

ROUND rounds (or truncates) to a specified number of fractional digits, but its return value is always normalized, removing trailing zeros. For example, ROUND(10.004,2) returns 10, not 10.00.

TRUNCATE truncates to a specified number of fractional digits. If the truncation results in trailing zeros, these trailing zeros are preserved. However, if *scale* is larger than the number of fractional decimal digits in the canonical form of *numeric-expr*, **TRUNCATE** does not zero-pad.

Use $JUSTIFY when rounding to a fixed number of fractional digits is important — for example, when representing monetary amounts. **$JUSTIFY** returns the specified number of trailing zeros following the rounding operation. When the number of digits to round is larger than the number of fractional digits, **$JUSTIFY** zero-pads. **$JUSTIFY** also right-aligns the numbers, so that the DecimalSeparator characters align in a column of numbers. **$JUSTIFY** does not truncate.

$DOUBLE Numbers

$DOUBLE IEEE floating point numbers are encoded using binary notation. Most decimal fractions cannot be exactly represented in this binary notation. When a **$DOUBLE** value is input to **ROUND** with a *scale* value and the rounding *flag* (*flag*=0, the default), the return value frequently contains more fractional digits than specified in *scale* because the fractional

decimal result is not representable in binary, so the return value must be rounded to the nearest representable **$DOUBLE** value, as shown in the following example:

SQL

```
SELECT ROUND(1234.5678,2),ROUND($DOUBLE(1234.5678),2)
```

If you are using **ROUND** to truncate a **$DOUBLE** value (*flag*=1), the return value for the **$DOUBLE** is truncated to the number of fractional digits specified by *scale*. The TRUNCATE function also truncates a **$DOUBLE** to the number of fractional digits specified by *scale*.

If you are using **ROUND** to round a **$DOUBLE** value and wish to return a specific *scale*, you should convert the $DOUBLE value to decimal representation before rounding the result.

ROUND with *flag*=0 (round, the default) returns `$DOUBLE("INF")` and `$DOUBLE("NAN")` as the empty string.

ROUND with *flag*=1 (truncate) returns `$DOUBLE("INF")` and `$DOUBLE("NAN")` as INF and NAN.

Arguments

numeric-expr

The number to be rounded. A numeric expression.

scale

An expression that evaluates to an integer that specifies the number of places to round to, counting from the decimal point. Can be zero, a positive integer, or a negative integer. If *scale* is a fractional number, InterSystems IRIS rounds it to the nearest integer.

flag

An optional boolean flag that specifies whether to round or truncate the *numeric-expr*: 0=round, 1=truncate. The default is 0.

Examples

The following example uses a *scale* of 0 (zero) to round several fractions to integers. It shows that 5 is always rounded up:

SQL

```
SELECT ROUND(5.99,0) AS RoundUp,
       ROUND(5.5,0) AS Round5,
       {fn ROUND(5.329,0)} AS Roundoff
```

The following example truncates the same fractional numbers as the previous example:

SQL

```
SELECT ROUND(5.99,0,1) AS Trunc1,
       ROUND(5.5,0,1) AS Trunc2,
       {fn ROUND(5.329,0,1)} AS Trunc3
```

The following **ROUND** functions round and truncate a negative fractional number:

SQL

```
SELECT ROUND(-0.987,2,0) AS Round1,
       ROUND(-0.987,2,1) AS Trunc1
```

The following example rounds off pi to four decimal digits:

SQL

```
SELECT {fn PI()} AS ExactPi, ROUND({fn PI()},4) AS ApproxPi
```

The following example specifies a *scale* larger than the number of decimal digits:

SQL

```
SELECT {fn ROUND(654.98700,9)} AS Rounded
```

it returns 654.987 (InterSystems IRIS removed the trailing zeroes before the rounding operation; no rounding or zero padding occurred).

The following example rounds off the value of Salary to the nearest thousand dollars:

SQL

```
SELECT Salary,ROUND(Salary, -3) AS PayBracket
FROM Sample.Employee
ORDER BY Salary
```

Note that if Salary is less than five hundred dollars, it is rounded to 0 (zero).

In the following example each **ROUND** specifies a negative *scale* as large or larger than the number to be rounded:

SQL

```
SELECT {fn ROUND(987,-3)} AS Round1,
       {fn ROUND(487,-3)} AS Round2,
       {fn ROUND(987,-4)} AS Round3,
       {fn ROUND(987,-5)} AS Round4
```

The first **ROUND** function returns 1000, because the rounded result has more digits than the *scale*. The other three **ROUND** functions return 0 (zero).

See Also

- $JUSTIFY function

- TRUNCATE function

- CEILING function

- FLOOR function

- MOD function

- ObjectScript functions: $DOUBLE, $NORMALIZE, $NUMBER

RPAD (SQL)

A string function that returns a string right-padded to a specified length.

Synopsis

```
RPAD(string-expression,length[,padstring])
```

Description

RPAD pads a string expression with trailing pad characters. It returns a copy of the string padded to *length* number of characters. If the string expression is longer than *length* number of characters, the return string is truncated to *length* number of characters.

If *string-expression* is NULL, **RPAD** returns NULL. If *string-expression* is the empty string (") **RPAD** returns a string consisting entirely of pad characters. The returned string is type VARCHAR.

RPAD can be used in queries against a linked table.

RPAD does not remove leading or trailing blanks; it pads the string including any leading or trailing blanks. To remove leading or trailing blanks before padding a string, use **LTRIM**, **RTRIM**, or **TRIM**.

Arguments

string-expression

A string expression, which can be the name of a column, a string literal, a host variable, or the result of another scalar function. Can be of any data type convertible to a VARCHAR data type. *string-expression* cannot be a stream.

length

An integer specifying the number of characters in the returned string.

padstring

An optional string consisting of a character or a string of characters used to pad the input *string-expression*. The *padstring* character or characters are appended to the right of *string-expression* to supply as many characters as need to create an output string of *length* characters. *padstring* may be a string literal, a column, a host variable, or the result of another scalar function. If omitted, the default is a blank space character.

Examples

The following example right pads column values with ^ characters (when needed) to return strings of length 16. Note that some Name strings are right padded, some Name strings are right truncated to return strings of length 16.

SQL

```
SELECT TOP 15 Name,RPAD(Name,16,'^') AS Name16
FROM Sample.Person
```

The following example right pads column values with the ^=^ pad string (when needed) to return strings of length 20. Note that the pad name string is repeated as many times as needed, and that some return strings contain partial pad strings:

SQL

```
SELECT TOP 15 Name,RPAD(Name,20,'^=^') AS Name20
FROM Sample.Person
```

See Also

- $JUSTIFY function
- LPAD function
- LTRIM function
- RTRIM function
- TRIM function

RTRIM (SQL)

A string function that returns a string with the trailing blanks removed.

Synopsis

```
RTRIM(string-expression)
```

```
{fn RTRIM(string-expression)}
```

Description

RTRIM strips the trailing blanks from a string expression, and returns the string as type VARCHAR. If *string-expression* is NULL, **RTRIM** returns NULL. If *string-expression* is a string consisting entirely of blank spaces, **RTRIM** returns the empty string ('').

RTRIM always returns data type VARCHAR, regardless of the data type of the input expression to be trimmed.

RTRIM leaves leading blanks; to remove leading blanks, use **LTRIM**. To remove leading and/or trailing characters of any type, use **TRIM**. To pad a string with trailing blanks or other characters, use **RPAD**. To create a string of blanks, use **SPACE**.

Note that **RTRIM** can be used as an ODBC scalar function (with the curly brace syntax) or as an SQL general function.

Arguments

string-expression

A string expression, which can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR).

Example

The following example removes the five trailing blanks from the string. It leaves the five leading blanks:

SQL

```
SELECT {fn RTRIM("     Test string with 5 leading and 5 trailing spaces.     ")}
```

Returns:

```
Before RTRIM
start:     Test string with 5 leading and 5 trailing spaces.     :end
After RTRIM
start:     Test string with 5 leading and 5 trailing spaces.:end
```

See Also

- LTRIM TRIM RPAD SPACE

SEARCH_INDEX (SQL)

A function that returns a set of values from the index's Find() method.

Synopsis

```
SEARCH_INDEX([[schema_name.]table-name.]index-name[,findparam[,...]])
```

Description

SEARCH_INDEX invokes the *index-name* **Find()** method and returns a set of values. You can optionally pass parameters to this **Find()** method. For example, SEARCH_INDEX(Sample.Person.NameIDX) invokes the Sample.Person.NameIDXFind() method.

SEARCH_INDEX can be used with the %FIND predicate in a **WHERE** clause to supply the OREF of an object that provides an abstract representation encapsulating a set of values. These values are commonly row IDs returned by a method called at query run time. **SEARCH_INDEX** invokes the index's **Find()** method to return this OREF.

The index must be found within the tables referenced by the SQL statement. An SQLCODE -151 error is generated if the specified *index-name* does not exist within the tables used by the SQL statement. An SQLCODE -152 error is generated if the specified *index-name* is not fully qualified, and is therefore ambiguous (could refer to more than one existing index) within the tables used by the SQL statement.

If the index exists, but it has no corresponding **Find()** method, a runtime SQLCODE -149 error is generated "SQL Function encountered an error", the error being <METHOD DOES NOT EXIST>.

For further details on the use of **SEARCH_INDEX**, refer to the SQL Search text search tool.

Arguments

table-name

An optional argument specifying the name of an existing table for which *index-name* is defined. Cannot be a view. The table's *schema_name* is optional. If omitted, all tables specified in the **FROM** clause are searched.

index-name

The index to be searched. The SqlName of the index map of an existing index.

findparam

An optional parameter or a comma-separated list of parameters to be passed to the index's **Find()** method.

Examples

The following example shows the usage of the %FIND predicate with **SEARCH_INDEX**:

SQL

```
SELECT Name FROM Sample.Person AS P
WHERE P.Name %FIND SEARCH_INDEX(Sample.Person.NameIDX)
```

See Also

- CREATE INDEX

- %FIND predicate

- %INSET predicate

- [Defining and Building Indexes](#)
- [Using Indexes](#)

SECOND (SQL)

A time function that returns the second for a datetime expression.

Synopsis

```
{fn SECOND(time-expression)}
```

Description

SECOND returns an integer from 0 to 59, and may return fractional seconds as well. The seconds are calculated for a $HOROLOG or $ZTIMESTAMP value, an ODBC format date string (with no time value), or a timestamp.

A *time-expression* timestamp can be either data type %Library.PosixTime (an encoded 64-bit signed integer), or data type %Library.TimeStamp (yyyy-mm-dd hh:mm:ss.fff).

To change the default time format, use the SET OPTION command.

You must supply either a timestamp string (yyyy-mm-dd hh:mm:ss) or a **$HOROLOG** string. A **$HOROLOG** string may be a full datetime string (63274,37279) or only the time integer portion of **$HOROLOG** (37279). You cannot supply a time string (hh:mm:ss); this always returns 0, regardless of the actual number of seconds.

The time portion of the datetime string must be a valid time. Otherwise, an SQLCODE -400 error <ILLEGAL VALUE> is generated. The seconds (ss) portion must be an integer in the range from 0 through 59. Leading zeros are optional on input; leading zeros are suppressed on output.

The date portion of the datetime string is not validated.

SECOND returns 0 seconds when the seconds portion is '0' or '00'. Zero seconds is also returned if an ODBC date with no time expression is supplied, or if the seconds portion of the time expression is omitted entirely ('hh', 'hh:mm', 'hh:mm:', or 'hh::').

The same time information can be returned using DATEPART or DATENAME.

This function can also be invoked from ObjectScript using the **SECOND()** method call:

```
$SYSTEM.SQL.Functions.SECOND(time-expression)
```

Fractional Seconds

SECOND returns fractions of a second if supplied in *time-expression*. Trailing zeros are truncated. If no fractional seconds are specified (for example: 38.00) the decimal separator is also truncated.

The standard InterSystems IRIS internal representation of time values ($HOROLOG) does not support fractional seconds. Timestamps do support fractional seconds.

The following SQL functions support fractional seconds: **SECOND**, **CURRENT_TIMESTAMP**, **DATENAME**, **DATEPART**, and **GETDATE**. **CURTIME**, **CURRENT_TIME**, and **NOW** do not support fractional seconds.

The SQL SET OPTION statement permits you to set the default precision (number of decimal digits) for fractional seconds.

The ObjectScript $ZTIMESTAMP special variable can be used to represent fractional seconds. The ObjectScript functions **$ZDATETIME**, **$ZDATETIMEH**, **$ZTIME**, and **$ZTIMEH** support fractional seconds.

Arguments

time-expression

An expression that is the name of a column, the result of another scalar function, or a string or numeric literal. It must resolve either to a timestamp string or a $HOROLOG string, where the underlying data type can be represented as %Time, %TimeStamp, or %PosixTime.

Examples

The following examples both return the number 38 because it is the thirty-eighth second of the time expression:

SQL

```
SELECT {fn SECOND('2018-02-16 18:45:38')} AS ODBCSeconds
```

SQL

```
SELECT {fn SECOND(67538)} AS HorologSeconds
```

The following example returns .9 seconds. The leading and trailing zeros are truncated:

SQL

```
SELECT {fn SECOND('2018-02-16 18:45:00.9000')} AS Seconds_Given
```

The following example returns 0 seconds because the seconds portion of the datetime string has been omitted:

SQL

```
SELECT {fn SECOND('2018-02-16 18:45')} AS Seconds_Given
```

The following example returns 0 seconds because the time expression has been omitted from the datetime string:

SQL

```
SELECT {fn SECOND('2018-02-16')} AS Seconds_Given
```

The following examples all return the seconds portion of the current time, in whole seconds:

SQL

```
SELECT {fn SECOND(CURRENT_TIME)} AS Sec_CurrentT,
       {fn SECOND({fn CURTIME()})} AS Sec_CurT,
       {fn SECOND({fn NOW()})} AS Sec_Now,
       {fn SECOND($HOROLOG)} AS Sec_Horolog,
       {fn SECOND($ZTIMESTAMP)} AS Sec_ZTS
```

The following example shows that leading zeros are suppressed. The first **SECOND** function returns a length 2, the others return a length of 1. An omitted time is considered to be 0 seconds, which has a length of 1:

SQL

```
SELECT LENGTH({fn SECOND('2018-02-15 11:45:22')}),
       LENGTH({fn SECOND('2018-02-15 03:05:06')}),
       LENGTH({fn SECOND('2018-02-15 3:5:6')}),
       LENGTH({fn SECOND('2018-02-15')})
```

The following example shows that the **SECOND** function recognizes the TimeSeparator character specified for the locale:

SQL

```
SELECT {fn SECOND('2018-02-16 18.45.38')}
```

See Also

- SQL concepts: Data Type, Date and Time Constructs

- SQL functions: HOUR, MINUTE, CURRENT_TIME, CURTIME, NOW, DATEPART, DATENAME

- ObjectScript function: $ZTIME

- ObjectScript special variables: $HOROLOG, $ZTIMESTAMP

SIGN (SQL)

A numeric function that returns the sign of a given numeric expression.

Synopsis

```
SIGN(numeric-expression)
```

```
{fn SIGN(numeric-expression)}
```

Description

SIGN returns the following:

- -1 if *numeric-expression* is less than zero.

- 0 (zero) if *numeric-expression* is zero: 0, +0, or -0.

- 1 if *numeric-expression* is greater than zero.

- NULL if *numeric-expression* is NULL, or if it is a non-numeric string.

SIGN can be used as either an ODBC scalar function (with the curly brace syntax) or as an SQL general function.

SIGN converts *numeric-expression* to canonical form before determining its value. For example, SIGN(-+-+3) and SIGN(-3+5) both return 1, indicating a positive number.

Note: In InterSystems SQL, two negative signs (hyphens) are the in-line comment indicator. For this reason, a **SIGN** argument specifying two successive negative signs must be presented as a numeric string enclosed in quotes.

Arguments

numeric-expression

A number for which the sign is to be returned.

Examples

The following examples shows the effects of **SIGN**:

SQL

```
SELECT SIGN(-49) AS PosNeg
```

returns -1.

SQL

```
SELECT {fn SIGN(-0.0)} AS PosNeg
```

returns 0.

SQL

```
SELECT SIGN(-+-16.748) AS PosNeg
```

returns 1.

SQL

```
SELECT {fn SIGN(NULL)} AS PosNeg
```

returns <null>.

See Also

- + (Positive) and – (Negative) unary operators

- ABS function

- ISNUMERIC function

- %PLUS and %MINUS collation functions

SIN (SQL)

A scalar numeric function that returns the sine, in radians, of an angle.

Synopsis

```
{fn SIN(numeric-expression)}
```

Description

SIN takes any numeric value and returns its sine as a floating point number. **SIN** returns NULL if passed a NULL value. **SIN** treats nonnumeric strings as the numeric value 0.

SIN returns a value with a precision of 19 and a scale of 18.

SIN can only be used as an ODBC scalar function (with the curly brace syntax).

You can use the DEGREES function to convert radians to degrees. You can use the RADIANS function to convert degrees to radians.

Arguments

numeric-expression

A numeric expression. This is an angle expressed in radians.

SIN returns either the NUMERIC or DOUBLE data type. If *numeric-expression* is data type DOUBLE, **SIN** returns DOUBLE; otherwise, it returns NUMERIC.

Example

The following example shows the effect of **SIN**:

SQL

```
SELECT {fn SIN(0.52)} AS Sine
```

returns 0.496880.

See Also

- SQL functions: ACOS, ASIN, ATAN, COS, COT, TAN
- ObjectScript function: $ZSIN

SPACE (SQL)

A string function that returns a string of spaces.

Synopsis

```
SPACE(count)
```

```
{fn SPACE(count)}
```

Description

SPACE returns a string of blank spaces *count* spaces long. If *count* is a numeric string, a decimal number, or a mixed numeric string, InterSystems IRIS resolves it to its integer portion. If *count* is a negative number or a nonnumeric string, InterSystems IRIS resolves it to 0.

To remove blank spaces from a string, use **LTRIM** (leading blanks) or **RTRIM** (trailing blanks).

Note: The **SPACE** function should not be confused with the SPACE collation type. SPACE collation prepends a single space to a value, forcing it to be evaluated as a string. To establish SPACE collation, CREATE TABLE provides a %SPACE collation keyword, and ObjectScript provides the **Collation()** method of the %SYSTEM.Util class.

Arguments

count

An integer expression specifying the number of blank spaces to return.

Examples

The following example returns a string of spaces the length of the *name* field:

SQL

```
SELECT SPACE(LENGTH(name))
    FROM Sample.Person
```

See Also

- LTRIM RTRIM TRIM

%SQLSTRING (SQL)

A collation function that sorts values as strings.

Synopsis

```
%SQLSTRING(expression[,maxlen])

%SQLSTRING expression
```

Description

%SQLSTRING converts *expression* to format that is sorted as a (case-sensitive) string. **%SQLSTRING** strips trailing whitespace (spaces, tabs, and so on) from the string, then adds one leading blank space to the beginning of the string. This appended blank space forces NULL and numeric values to be collated as strings. Leading and trailing zeros are removed from numbers.

Because **%SQLSTRING** appends a blank space to all values, it collates a NULL value as a blank space, with a string length of 1. **%SQLSTRING** collates any value containing only whitespace (spaces, tabs, and so on) as the SQL empty string ("). When **%SQLSTRING** appends a blank space to an empty (zero-length) string, it collates as a blank space plus the internal representation of an empty string, $CHAR(0), resulting in a string length of 2.

The optional *maxlen* argument truncates the *expression* string to the specified number of characters when indexing or collating. For example, if you insert a string with *maxlen* truncation, the full string is inserted and can be retrieved by a **SELECT** statement; the index global for this string is truncated to the specified length. This means that **ORDER BY** and comparison operations only evaluate the truncated index string. Such truncation is especially useful for indexing on strings that exceed the maximum character length for InterSystems IRIS subscripts. With the *maxlen* argument, if you need to index on a long field, you can use the truncation length parameter.

%SQLSTRING performs *maxlen* truncation after converting *expression*; if *maxlen* exceeds the length of the converted *expression* no padding is added. Note that within InterSystems IRIS, no string can exceed the string length limit. No maximum is enforced for *maxlen* explicitly but InterSystems IRIS will issue a <MAXSTRING> error if applicable.

You can perform the same collation conversion in ObjectScript using the **Collation()** method of the %SYSTEM.Util class:

ObjectScript

```
WRITE $SYSTEM.Util.Collation("The quick, BROWN fox.",8)
```

This function can also be invoked from ObjectScript using the **SQLSTRING()** method call:

ObjectScript

```
WRITE $SYSTEM.SQL.Functions.SQLSTRING("The quick, BROWN fox.")
```

Both of these methods support truncation after SQLSTRING conversion. Note that the truncation length must include the appended blank:

ObjectScript

```
WRITE $SYSTEM.Util.Collation("The quick, BROWN fox.",8,6),!
WRITE $SYSTEM.SQL.SQLSTRING("The quick, BROWN fox.",6)
```

For a not case-sensitive string conversion, refer to %SQLUPPER.

Note: To change the system-wide default collation from %SQLUPPER (which is not case-sensitive) to %SQLSTRING (which is case-sensitive), use the following command:

ObjectScript

```
WRITE $$SetEnvironment^%apiOBJ("collation","%Library.String","SQLSTRING")
```

After issuing this command, you must purge indexes, recompile all classes, then rebuild indexes. Do not rebuild indexes while the table's data is being accessed by other users. Doing so may result in inaccurate query results.

Arguments

expression

A string expression, which can be the name of a column, a string literal, or the result of another function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR). *expression* can be a subquery.

maxlen

An optional date or timestamp expression from which the day of the month value is to be returned. An expression that is the name of a column, the result of another scalar function, or a date or timestamp literalA positive integer, which specifies that the collated value will be truncated to the value of *maxlen*. Note that *maxlen* includes the appended leading blank space. You can enclose *maxlen* with double parentheses to suppress literal substitution: ((*maxlen*)).

Examples

The following query uses **%SQLSTRING** in the **WHERE** clause to perform a case-sensitive select:

SQL

```
SELECT Name FROM Sample.Person
WHERE %SQLSTRING Name %STARTSWITH %SQLSTRING 'Al'
ORDER BY Name
```

By default, **%STARTSWITH** string comparisons are not case-sensitive. This example uses the **%SQLSTRING** format to make this comparison case-sensitive. It returns all names that begin with "Al" (such as Allen, Alton, etc.). Note when using **%STARTSWITH**, you should apply **%SQLSTRING** collation to both sides of the statement.

The following example uses **%SQLSTRING** with a string truncation to return the first two characters of each name. Note that the string truncation is 3 (not 2) because of the leading blank added by **%SQLSTRING**. The **ORDER BY** clause uses this two-character field to put the rows in a rough collation sequence:

SQL

```
SELECT Name, %SQLSTRING(Name,3) AS FirstTwo
FROM Sample.Person
ORDER BY FirstTwo
```

This example returns the truncated values without changing the case of letters.

The following example applies **%SQLSTRING** to a subquery:

SQL

```
SELECT TOP 5 Name, %SQLSTRING((SELECT Name FROM Sample.Company),10) AS Company
FROM Sample.Person
```

See Also

- CREATE TABLE

- **%STARTSWITH** predicate

- **%SQLUPPER** collation function

- **%TRUNCATE** collation function

- Collation

%SQLUPPER (SQL)

A collation function that sorts values as uppercase strings.

Synopsis

```
%SQLUPPER(expression[,maxlen])
%SQLUPPER expression
```

Description

SQLUPPER is the default collation.

%SQLUPPER converts *expression* to a format that is sorted as a (not case-sensitive) uppercase string. **%SQLUPPER** converts all alphabetic characters to uppercase, strips trailing whitespace (spaces, tabs, and so on) from the string, then adds one leading blank space to the beginning of the string. This prepended blank space causes NULL and numeric values to be collated as strings.

SQL converts numeric values to canonical form (removing leading and trailing zeros, expanding exponents, etc.) before passing the number to the function. SQL does not convert numeric strings to canonical form.

Because **%SQLUPPER** prepends a blank space to all values, it collates a NULL value as a blank space, with a string length of 1. **%SQLUPPER** collates any value containing only whitespace (spaces, tabs, and so on) as the SQL empty string (''). When **%SQLUPPER** prepends a blank space to an empty (zero-length) string, it collates as a blank space plus the internal representation of an empty string, $CHAR(0), resulting in a string length of 2.

The optional *maxlen* argument truncates the converted *expression* string to the specified number of characters when indexing or collating. For example, if you insert a string with *maxlen* truncation, the full string is inserted and can be retrieved by a **SELECT** statement; the index global for this string is truncated to the specified length. This means that **ORDER BY** and comparison operations only evaluate the truncated index string. Such truncation is especially useful for indexing on strings that exceed the maximum character length for InterSystems IRIS subscripts. With the *maxlen* argument, if you need to index on a long field, you can use the truncation length parameter.

%SQLUPPER performs *maxlen* truncation after converting *expression*; if *maxlen* exceeds the length of the converted *expression* no padding is added. Note that within InterSystems IRIS, no string can exceed the string length limit. No maximum is enforced for *maxlen* explicitly but InterSystems IRIS will issue a <MAXSTRING> error if applicable.

You can perform the same collation conversion in ObjectScript using the **Collation()** method of the %SYSTEM.Util class:

ObjectScript

```
WRITE $SYSTEM.Util.Collation("The quick, BROWN fox.",7)
```

This function can also be invoked from ObjectScript using the **SQLUPPER()** method call:

ObjectScript

```
WRITE $SYSTEM.SQL.Functions.SQLUPPER("The quick, BROWN fox.")
```

Both of these methods support truncation after SQLUPPER conversion. Note that the truncation length must include the prepended blank:

ObjectScript

```
WRITE $SYSTEM.Util.Collation("The quick, BROWN fox.",7,6),!
WRITE $SYSTEM.SQL.SQLUPPER("The quick, BROWN fox.",6)
```

For a case-sensitive string conversion, refer to %SQLSTRING.

Note: To change the system-wide default collation from %SQLUPPER (which is not case-sensitive) to %SQLSTRING (which is case-sensitive), use the following command:

ObjectScript

```
WRITE $$SetEnvironment^%apiOBJ("collation","%Library.String","SQLSTRING")
```

After issuing this command, you must purge indexes, recompile all classes, then rebuild indexes. Do not rebuild indexes while the table's data is being accessed by other users. Doing so may result in inaccurate query results.

Other Case Conversion Functions

The **%SQLUPPER** function is the preferred way in SQL to convert a data value for not case-sensitive comparison or collation. **%SQLUPPER** adds a leading blank space to the beginning of the data, which forces numeric data and the NULL value to be interpreted as strings.

The following are other functions for converting the case of a data value:

*   UPPER and UCASE: converts letters to uppercase, has no effect on number characters, punctuation characters, embedded spaces, and leading and trailing blank spaces. Does not force numerics to be interpreted as a string.

*   LOWER and LCASE: converts letters to lowercase, has no effect on number characters, punctuation characters, embedded spaces, and leading and trailing blank spaces. Does not force numerics to be interpreted as a string.

*   %SQLSTRING: does not convert letter case. However, it adds a leading blank space to the beginning of the data, which forces numeric data and the NULL value to be interpreted as strings.

Alphanumeric Collation Order

The case conversion functions collate data values that begin with a number using different algorithms, as follows:

%MVR	%SQLUPPER, %SQLSTRING, and all other case conversion functions
6 Oak Avenue, 66 Main Street, 66 Oak Street, 641 First Place, 665 Ash Drive, 672 Main Court, 709 Oak Avenue, 5988 Clinton Avenue, 6023 Washington Court, 6090 Elm Court, 6185 Clinton Drive, 6209 Clinton Street, 6284 Oak Drive, 6310 Franklin Street, 6406 Maple Place, 6572 First Avenue, 6643 First Street, 6754 Oak Court, 6986 Madison Blvd, 7000 Ash Court,	5988 Clinton Avenue, 6 Oak Avenue, 6023 Washington Court, 6090 Elm Court, 6185 Clinton Drive, 6209 Clinton Street, 6284 Oak Drive, 6310 Franklin Street, 6406 Maple Place, 641 First Place, 6572 First Avenue, 66 Main Street, 66 Oak Street, 6643 First Street, 665 Ash Drive, 672 Main Court, 6754 Oak Court, 6986 Madison Blvd, 7000 Ash Court, 709 Oak Avenue,

Arguments

expression

A string expression, which can be the name of a column, a string literal, or the result of another function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR). *expression* can be a subquery.

maxlen

An optional integer, which specifies that the collated value will be truncated to the value of *maxlen*. Note that *maxlen* includes the prepended leading blank space. You can enclose *maxlen* with double parentheses to suppress literal substitution: ((*maxlen*)).

Examples

The following query uses **%SQLUPPER** with a string truncation to return the first two characters of each name in uppercase. Note that the string truncation is 3 (not 2) because of the leading blank added by **%SQLUPPER**. The **ORDER BY** clause uses this two-character field to put the rows in a rough collation sequence:

SQL

```
SELECT Name, %SQLUPPER(Name,3) AS FirstTwo
FROM Sample.Person
ORDER BY FirstTwo
```

The following example applies **%SQLUPPER** to a subquery:

SQL

```
SELECT TOP 5 Name, %SQLUPPER((SELECT Name FROM Sample.Company),10) AS Company
FROM Sample.Person
```

See Also

- CREATE TABLE

- %STARTSWITH predicate

- %SQLSTRING collation function

- %TRUNCATE collation function

- Collation

SQRT (SQL)

A numeric function that returns the square root of a given numeric expression.

Synopsis

```
SQRT(numeric-expression)
```

```
{fn SQRT(numeric-expression)}
```

Description

SQRT returns the square root of *numeric-expression*. The *numeric-expression* must be a positive number. A negative *numeric-expression* (other than -0) generates an SQLCODE -400 error. **SQRT** returns NULL if passed a NULL value.

SQRT returns a value with a precision of 36 and a scale of 18.

SQRT can be specified as a regular scalar function or as an ODBC scalar function (with the curly brace syntax).

Arguments

numeric-expression

An expression that resolves to a positive number from which the square root is calculated.

SQRT returns either the NUMERIC or DOUBLE data type. If *numeric-expression* is data type DOUBLE, **SQRT** returns DOUBLE; otherwise, it returns NUMERIC.

Examples

The following example shows the two **SQRT** syntax forms. Both return the square root of 49:

SQL

```
SELECT SQRT(49) AS SRoot,{fn SQRT(49)} AS ODBCSRoot
```

The following embedded SQL example returns the square roots of the integers 0 through 10:

ObjectScript

```
SET a=0
WHILE a<11 {
&sql(SELECT SQRT(:a) INTO :b)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE
  QUIT }
ELSE {
  WRITE !,"The square root of ",a," = ",b
  SET a=a+1 }
}
```

See Also

- SQL functions: POWER ROUND SQUARE

- ObjectScript function: $ZSQR

SQUARE (SQL)

A scalar numeric function that returns the square of a number.

Synopsis

```
SQUARE(numeric-expression)
```

Description

SQUARE returns the square of *numeric-expression*. **SQUARE** returns NULL if passed a NULL value.

The precision and scale returned by **SQUARE** are the same as those returned by the SQL multiplication operator.

Arguments

numeric-expression

An expression that resolves to a numeric value.

SQUARE returns either the NUMERIC or DOUBLE data type. If *numeric-expression* is data type DOUBLE, **SQUARE** returns DOUBLE; otherwise, it returns NUMERIC.

Examples

The following Embedded SQL example returns the squares of the integers 0 through 10:

ObjectScript

```
SET a=0
WHILE a<11 {
&sql(SELECT SQUARE(:a) INTO :b)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE
  QUIT }
ELSE {
  WRITE !,"The square of ",a," = ",b
  SET a=a+1 }
}
```

See Also

- SQL functions: POWER, ROUND, SQRT

- ObjectScript function: $ZPOWER

STR (SQL)

A function that converts a numeric to a string.

Synopsis

```
STR(number[,length[,decimals]])
```

Description

STR converts a numeric to the STRING format, truncating the numeric based on the values of *length* and *decimals*. The *length* argument must be large enough to include the entire integer portion of the number, and, if *decimals* is specified, that number of decimal digits plus 1 (for the decimal point). If *length* is not large enough, **STR** returns a string of asterisks (*) equal to *length*.

STR converts numerics to their canonical form before string conversion. It therefore performs arithmetic operations, removes leading and trailing zeros and leading plus signs from numbers.

If the *number* argument is NULL, **STR** returns NULL. If the *number* argument is the empty string ("), **STR** returns the empty string. **STRING** retains whitespace.

Arguments

number

An expression that resolves to a numeric. It can be a field name, a numeric, or the result of another function. If a field name is specified, the logical value is used.

length

An optional integer specifying the total length of the desired output string, including all characters (digits, decimal point, sign, blank spaces). The default is 10.

decimals

An optional integer specifying the number of places to the right of the decimal point to include. The default is 0.

Example

In the following example, **STR** converts numerics into a string:

SQL

```
SELECT STR(123),
    STR(123,4),
    STR(+00123.45,3),
    STR(+00123.45,3,1),
    STR(+00123.45,5,1)
```

The first **STR** function returns a string consisting of 7 leading blanks and the number 123; the seven leading blanks are because the default string length is 10. The second **STR** function returns the string " 123"; note the leading blank needed to return a string of length 4. The third **STR** function returns the string "123"; the numeric is put into canonical form, and *decimals* defaults to 0. The fourth **STR** function returns "***" because the string length is not long enough to encompass the entire number as specified; the number of asterisks indicates the string length. The fifth **STR** function returns "123.4"; note that the *length* must be 5 to include the decimal digit.

See Also

- STRING, %SQLUPPER, %SQLSTRING

STRING (SQL)

A function that converts and concatenates expressions into a string.

Synopsis

```
STRING(string1[,string2][,...][,stringN])
```

Description

STRING converts one or more *string*s to the STRING format, and then concatenates these strings into a single string. No case transformation is performed.

STRING converts numerics to their canonical form before string conversion. It therefore performs arithmetic operations, removes leading and trailing zeros and leading plus signs from numbers.

If any of the *string* arguments is NULL or the empty string (''), **STRING** concatenates all other arguments and removes NULL and the empty string from the concatenation. If all of the *string* arguments are NULL, **STRING** returns NULL. If all of the *string* arguments are the empty string (''), **STRING** returns the empty string. **STRING** retains whitespace.

You can use the %SQLSTRING function to convert a data value for case-sensitive string comparison, or the %SQLUPPER function to convert a data value for not case-sensitive string comparison.

Arguments

string

An expression, which can be a field name, a string literal, a numeric, or the result of another function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR). If a field name is specified, the logical value is used.

Examples

In the following example, **STRING** concatenates three substrings into a single string. The example shows the handling of blank spaces, the empty string, and NULL:

SQL

```
SELECT STRING('a','b','c'),
    STRING('a',' ','c'),
    STRING('a','','c'),
    STRING('a',NULL,'c')
```

In the following example, **STRING** converts numerics into a string. All of these **STRING** functions return the string '123':

SQL

```
SELECT STRING(123),
    STRING(+00123.00),
    STRING('1',23),
    STRING(1,(10*2)+3)
```

In the following example, **STRING** retrieves sample data from fields and concatenates it into a string:

SQL

```
SELECT STRING(Name,Age)
FROM Sample.Person
```

See Also

- %SQLUPPER, %SQLSTRING, STR

STUFF (SQL)

A string function that replaces a substring within a string.

Synopsis

```
STUFF(string,start,length,substring)
```

Description

STUFF replaces a substring with another substring. It identifies the substring to be replaced by location and length, and replaces it with *substring*.

This function provides compatibility with Transact-SQL implementations.

The replacement *substring* may be longer or shorter than the original value. To delete the original value, *substring* can be the empty string ('').

The *start* value must be within the current length of *string*. You can append a *substring* to the beginning of *string* by specifying a *start* value of 0. The empty string or a nonnumeric value is treated as 0.

Specifying NULL for the *start*, *length*, or *substring* argument returns NULL.

STUFF cannot use a %Stream.GlobalCharacter field for either the *string* or *substring* argument. Attempting to do so generates an SQLCODE -37 error.

REPLACE and STUFF

Both **REPLACE** and **STUFF** perform substring replacement. **REPLACE** searches for a substring by data value. **STUFF** searches for a substring by string position and length.

For a list of functions that search for a substring, refer to String Manipulation.

Arguments

string

A string expression that is the target for the substring replacement.

start

The starting point for replacement, specified as a positive integer. A character count from the beginning of *string*, counting from 1. Permitted values are 0 through the length of *string*. To append characters, specify a *start* of 0 and a *length* of 0. The empty string or a nonnumeric value is treated as 0.

length

The number of characters to replace, specified as a positive integer. To insert characters, specify a *length* of 0. To replace all characters after *start*, specify a *length* greater than the number of existing characters. The empty string or a nonnumeric value is treated as 0.

substring

A string expression used to replace the substring identified by its starting point and length. Can be longer or shorter than the substring it replaces. Can be the empty string.

Examples

The following example shows a single-character substitution, turning BOLT into BELT:

SQL

```
SELECT STUFF('BOLT',2,1,'E')
```

The following examples replace an 8-character substring (Kentucky) with a longer 12-character substring and a shorter 2-character substring:

SQL

```
SELECT STUFF('In my old Kentucky home',11,8,'Rhode Island'),
       STUFF('In my old Kentucky home',11,8,'KY')
```

The following example inserts a substring:

SQL

```
SELECT STUFF('In my old Kentucky home',19,0,' (KY)')
```

The following example appends a substring to the beginning of the string:

SQL

```
SELECT STUFF('In my old Kentucky home',0,0,'The sun shines bright ')
```

The following example deletes an 8-character substring by replacing it with the empty string:

SQL

```
SELECT STUFF('In my old Kentucky home',11,8,'')
```

See Also

- REPLACE function
- $EXTRACT function
- SUBSTRING function
- SUBSTR function
- String Manipulation

SUBSTR (SQL)

A string function that returns a substring derived from a specified string expression.

Synopsis

```
SUBSTR(string-expression,start[,length])
```

Arguments

Argument	Description
string-expression	The string expression from which the substring is to be derived. The expression can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR).
start	An integer that specifies where in *string-expression* the substring will begin. A positive starting position specifies the number of characters from the beginning of the string. The first character in *string-expression1* is at position 1. A negative starting position specifies the number of characters from the end of the string. If *start* is 0 (zero), it is treated as 1.
length	*Optional* — A positive integer that specifies the length of the substring to return. This value specifies that the substring ends *length* characters to the right of the starting position. If omitted, substring goes from *start* to the end of *string-expression*. If *length* is 0 or a negative number, InterSystems IRIS returns NULL.

Description

Because *start* can be negative, you can obtain a substring from either the beginning or end of the original string.

Floating-point numbers passed as arguments to **SUBSTR** are converted to integers by truncating the fractional portion.

- If *start* is 0, –0, or 1, the returned substring begins with the first character of the string.

- If *start* is a negative number the returned substring begins that number of characters from the end of the string, with -1 representing the last character of the string. If the negative number is so large that its value counted backwards from the end of the string would position before the beginning of the string, the returned substring begins with the first character of the string.

- If *start* is past the end of the string, NULL is returned.

- If *length* larger than the remaining characters in the string, the substring from *start* to the end of the string is returned.

- If *length* is less than 1, NULL is returned.

- If either *start* or *length* is NULL, NULL is returned.

SUBSTR cannot be used with stream data. If *string-expression* is a stream field, **SUBSTR** generates an SQLCODE -37. Use **SUBSTRING** to extract a substring from stream data.

SUBSTR is supported for Oracle compatibility.

Examples

The following example returns the substring CDEFG because it specifies that the substring begin at the third character (C) and continue to the end of the string:

SQL

```
SELECT SUBSTR('ABCDEFG',3) AS Sub
```

The following example returns the substring CDEF because it specifies that the substring begin at the third character (C) and continue for four characters (until F):

SQL

```
SELECT SUBSTR('ABCDEFG',3,4) AS Sub
```

The following example returns the substring CDEF because it specifies that InterSystems IRIS should first count five characters backwards from the end of the original string, and then return the next four characters:

SQL

```
SELECT SUBSTR('ABCDEFG',-5,4) AS Sub
```

See Also

- SQL function: SUBSTRING

- ObjectScript functions: $EXTRACT $PIECE

SUBSTRING (SQL)

A string function that returns a substring from data of any data type, including stream data.

Synopsis

```
SUBSTRING(string-expression,start[,length])
SUBSTRING(string-expression FROM start [FOR length])
{fn SUBSTRING(string-expression,start[,length])}
```

Arguments

Argument	Description
string-expression	The string expression from which the substring is to be derived. An expression, which can be the name of a column, a string literal, or the result of another scalar function. This field can be of any data type: a string (such as CHAR or VARCHAR), a numeric, or a data stream field of data type %Stream.GlobalCharacter or %Stream.GlobalBinary.
start	An integer that specifies the position in *string-expression* to begin the substring. The first character in *string-expression* is at position 1. If the start position is higher than the length of the string, **SUBSTRING** returns an empty string (''). If the start position is lower than 1 (zero, or a negative number) the substring begins at position 1, but the *length* of the substring is reduced by the start position.
length	Optional — An integer that specifies the length of the substring to return. If *length* is not specified, the default is to return the rest of the string.

Description

SUBSTRING takes data of any data type and returns a substring of that data as data type %String. The substring can, of course, be the full data value returned as a string.

The value of *start* controls the starting point of the substring:

- If *start* is 1, the substring begins at the beginning of *string-expression*.

- If *start* is greater than 1, the substring begins at that character position counted from the beginning of *string-expression*.

- If *start* is less than 1, the substring begins at the beginning of *string-expression*, but the value of *length* is decremented by a corresponding amount. Thus, if *start* is 0, the value of *length* is diminished by 1; if *start* is –1, the value of *length* is diminished by 2.

The value of *length* controls the size of the substring:

- If *length* is a positive value (1 or greater), the substring ends *length* number of characters to the right of the *start* position. (This effective length may be diminished if the *start* number is less than 1.)

- If *length* is larger than the number of character remaining in the string, all characters to the right of the starting position through the end of *string-expression* are returned.

- If *length* is zero, NULL is returned.

- If *length* is a negative number, InterSystems IRIS issues an SQLCODE –140 error.

SUBSTRING can be used as an ODBC scalar function (with the curly brace syntax) or as an SQL general function.

Return Value

If *string-expression* is any %String data type, the **SUBSTRING** return value is the same data type as the *string-expression* data type. This allows **SUBSTRING** to handle user-defined string data types with special encoding.

If *string-expression* is *not* a %String data type (for example, %Stream.GlobalCharacter), the **SUBSTRING** return value is %String.

If any **SUBSTRING** argument value is NULL, **SUBSTRING** returns NULL.

Stream Data

Unlike most SQL string functions, **SUBSTRING** can be used with stream data. The *string-expression* can be a field of data type %Stream.GlobalCharacter or %Stream.GlobalBinary. **SUBSTRING** returns the extracted subset of the stream data as %String data type. If *start* is 1 and *length* is omitted, **SUBSTRING** returns the full stream data value as a %String.

SUBSTRING can therefore be used to supply character stream data as a string to other SQL string functions. The following example uses **SUBSTRING** to allow **CHARINDEX** to search the first 1000 characters of a %Stream.GlobalCharacter field containing DNA nucleotide sequences for the first occurrence of the substring TTAGGG and returns that position as an integer:

SQL

```
SELECT CHARINDEX('TTAGGG',SUBSTRING(DNASeq,1,1000)) FROM Sample.DNASequences
```

SUBSTRING vs. SUBSTR

- **SUBSTRING** extracts a substring from a *start* position counted from the beginning of a *string-expression*. SUBSTR can extract a substring from either the beginning or the end of a string.

- **SUBSTRING** can be used with stream data; **SUBSTR** cannot be used with stream data.

Examples

This example returns the string "forward":

SQL

```
SELECT {fn SUBSTRING( 'forward pass',1,7 )} AS SubText
```

This example returns the string "pass":

SQL

```
SELECT {fn SUBSTRING( 'forward pass',9,4 )} AS SubText
```

The following example returns the first four characters of each name:

SQL

```
SELECT Name,SUBSTRING(Name,1,4) AS FirstFour
FROM Sample.Person
```

The following example demonstrates another syntactical form of **SUBSTRING**. This example is functionally the same as the previous example:

SQL

```
SELECT Name,SUBSTRING(Name FROM 1 FOR 4) AS FirstFour
FROM Sample.Person
```

The following example shows how the *length* is reduced by a *start* value of less than 1. (A *start* value of 0 reduces *length* by 1, a *start* value of -1 reduces *length* by 2, and so forth.) In this case, *length* is reduced by 3, so only one character ("A") is returned:

SQL

```
SELECT {fn SUBSTRING( 'ABCDEFG',-2,4 )} AS SubText
```

See Also

- SQL function: SUBSTR

- ObjectScript functions: $EXTRACT, $PIECE

SYSDATE (SQL)

A date/time function that returns the current local date and time.

Synopsis

```
SYSDATE
```

Description

SYSDATE takes no arguments and returns the current local date and time as a timestamp in either %TimeStamp data type format (yyyy-mm-dd hh:mm:ss.ffff) or %PosixTime data type format (an encoded 64-bit signed integer). **SYSDATE** returns the current local date and time for this timezone; it adjusts for local time variants, such as Daylight Saving Time.

By default, **SYSDATE** returns time in whole second increments. This default can be configured.

Note: **SYSDATE** is a synonym for the argumentless CURRENT_TIMESTAMP function. The **CURRENT_TIMESTAMP** function is preferred for use in InterSystems SQL. The **SYSDATE** function is provided for compatibility with other versions of SQL.

See Also

- CURRENT_TIMESTAMP

%SYSTEM_SQL.DefaultSchema()

A function that returns the default schema for the current process in the current namespace.

Synopsis

```
%SYSTEM_SQL.DefaultSchema()
```

Description

%SYSTEM_SQL.DefaultSchema() takes no arguments. It returns the default schema for the current process in the current namespace. Argument parentheses are required.

See Also

- **Class reference**

TAN (SQL)

A scalar numeric function that returns the tangent, in radians, of an angle.

Synopsis

```
{fn TAN(numeric-expression)}
```

Description

TAN takes any numeric value and returns its tangent. **TAN** returns NULL if passed a NULL value. **TAN** treats nonnumeric strings as the numeric value 0.

TAN returns a value with a precision of 36 and a scale of 18.

TAN can only be used as an ODBC scalar function (with the curly brace syntax).

You can use the DEGREES function to convert radians to degrees. You can use the RADIANS function to convert degrees to radians.

Arguments

numeric-expression

A numeric expression. This is an angle expressed in radians.

TAN returns either the NUMERIC or DOUBLE data type. If *numeric-expression* is data type DOUBLE, **TAN** returns DOUBLE; otherwise, it returns NUMERIC.

Example

The following example shows the effect of **TAN**.

SQL

```
SELECT {fn TAN(0.52)} AS Tangent
```

returns 0.572561.

See Also

- SQL functions: ACOS, ASIN, ATAN, COS, COT, SIN
- ObjectScript function: $ZTAN

TIMESTAMPADD (SQL)

A scalar date/time function that returns a new timestamp calculated by adding a number of intervals of a specified date part to a timestamp.

Synopsis

```
{fn TIMESTAMPADD(interval-type,integer-exp,timestamp-exp)}
```

Arguments

Argument	Description
interval-type	The type of time/date interval that *integer-exp* represents, specified as a keyword.
integer-exp	An integer value expression that is to be added to *timestamp-exp*.
timestamp-exp	A timestamp value expression, which will be increased by the value of *integer-exp*.

Description

The **TIMESTAMPADD** function modifies a date/time expression by incrementing the specified date part by the specified number of units. For example, if *interval-type* is SQL_TSI_MONTH and *integer-exp* is 5, **TIMESTAMPADD** increments *timestamp-exp* by five months. You can also decrement a date part by specifying a negative integer for *integer-exp*.

TIMESTAMPADD returns a timestamp of the same data type as the input *timestamp-exp*. This timestamp can be in either %Library.TimeStamp data type format (yyyy-mm-dd hh:mm:ss.ffff) or %Library.PosixTime data type format (an encoded 64-bit signed integer).

Note that **TIMESTAMPADD** can only be used as an ODBC scalar function (with the curly brace syntax).

Similar time/date modification operations can be performed on a timestamp using the DATEADD general function.

Interval Types

The *interval-type* argument can be one of the following timestamp intervals:

*   SQL_TSI_FRAC_SECOND
*   SQL_TSI_SECOND
*   SQL_TSI_MINUTE
*   SQL_TSI_HOUR
*   SQL_TSI_DAY
*   SQL_TSI_WEEK
*   SQL_TSI_MONTH
*   SQL_TSI_QUARTER
*   SQL_TSI_YEAR

These timestamp intervals may be specified with or without enclosing quotation marks, using single quotes or double quotes. They are not case-sensitive.

Incrementing or decrementing a timestamp interval causes other intervals to be modified appropriately. For example, incrementing the hour past midnight automatically increments the day, which may in turn increment the month, and so forth. **TIMESTAMPADD** always returns a valid date, taking into account the number of days in a month, and calculating

for leap year. For example, incrementing January 31 by one month returns February 28 (the highest valid date in the month), unless the specified year is a leap year, in which case it returns February 29.

You can increment or decrement by fractional seconds of three digits of precision. Specify fractional seconds as an integer count of thousandths of a second (001 through 999).

DATEADD and **TIMESTAMPADD** handle quarters (3-month intervals); **DATEDIFF** and **TIMESTAMPDIFF** do not handle quarters.

%TimeStamp Format

If the *timestamp-exp* argument is in %Library.TimeStamp data type format (yyyy-mm-dd hh:mm:ss.ffff) the following rules apply:

- If *timestamp-exp* specifies only a time value, the date portion of *timestamp-exp* is set to '1900–01–01' before calculating the resulting timestamp.

- If *timestamp-exp* specifies only a date value, the time portion of *timestamp-exp* is set to '00:00:00' before calculating the resulting timestamp.

- The *timestamp-exp* can include or omit fractional seconds. The *timestamp-exp* can include any number of digits of precision, but *interval-type* SQL_TSI_FRAC_SECOND specifies exactly three digits of precision. Attempting to specify a SQL_TSI_FRAC_SECOND of less than or more than three digits can have unpredictable results.

Range and Value Checking

TIMESTAMPADD performs the following checks on %Library.TimeStamp input values:

- All specified parts of the *timestamp-exp* must be valid before any **TIMESTAMPADD** operation can be performed.

- A date string must be complete and properly formatted with the appropriate number of elements and digits for each element, and the appropriate separator character. Years must be specified as four digits. An invalid date value results in an SQLCODE -400 error.

- Date values must be within a valid range. Years: 0001 through 9999. Months: 1 through 12. Days: 1 through 31. Hours: 00 through 23. Minutes: 0 through 59. Seconds: 0 through 59. The number of days in a month must match the month and year. For example, the date '02–29' is only valid if the specified year is a leap year. An invalid date value results in an SQLCODE -400 error.

- The incremented (or decremented) year value returned must be within the range 0001 through 9999. Incrementing or decrementing beyond this range returns <null>.

- Date values less than 10 may include or omit a leading zero. Other non-canonical integer values are not permitted. Therefore, a Day value of '07' or '7' is valid, but '007', '7.0' or '7a' are not valid. Date values less than 10 are always returned with a leading zero.

- Time values may be wholly or partially omitted. If *timestamp-exp* specifies an incomplete time, zeros are supplied for the unspecified parts.

- An hour value less than 10 must include a leading zero. Omitting this leading zero results in an SQLCODE -400 error.

Examples

The following example adds 1 week to the original timestamp:

SQL

```
SELECT {fn TIMESTAMPADD(SQL_TSI_WEEK,1,'2017–12–20 12:00:00')}
```

returns 2017-12-27 12:00:00, because adding 1 week adds 7 days.

The following example adds 5 months to the original timestamp:

SQL

```
SELECT {fn TIMESTAMPADD(SQL_TSI_MONTH,5,'2017-12-20 12:00:00')}
```

returns 2018-05-20 12:00:00 because in this case adding 5 months also increments the year.

The following example also adds 5 months to the original timestamp:

SQL

```
SELECT {fn TIMESTAMPADD(SQL_TSI_MONTH,5,'2018-01-31 12:00:00')}
```

returns 2018-06-30 12:00:00. Here **TIMESTAMPADD** modified the day value as well as the month, because simply incrementing the month would result in June 31, which is an invalid date.

The following example increments the original timestamp by 45 minutes:

SQL

```
SELECT {fn TIMESTAMPADD(SQL_TSI_MINUTE,45,'2017-12-20 00:00:00')}
```

returns 2017-12-20 00:45:00.

The following example decrements the original timestamp by 45 minutes:

SQL

```
SELECT {fn TIMESTAMPADD(SQL_TSI_MINUTE,-45,'2017-12-20 00:00:00')}
```

returns 2017-12-19 23:15:00. Note that in this case decrementing the time also decremented the day.

See Also

- TIMESTAMPDIFF, DATEADD, DATENAME, DATEPART, TO_POSIXTIME, TO_TIMESTAMP

TIMESTAMPDIFF (SQL)

A scalar date/time function that returns an integer count of the difference between two timestamps for a specified date part.

Synopsis

```
{fn TIMESTAMPDIFF(interval,startDate,endDate)}
```

Description

- **{fn TIMESTAMPDIFF(*interval*,*startDate*,*endDate*)}** returns the difference between the starting and ending timestamps (*startDate* minus *endDate*) for the specified date part interval (seconds, days, weeks, and so on). The function returns an INTEGER value representing the number of intervals between the two timestamps. If *endDate* is earlier than *startDate*, **TIMESTAMPDIFF** returns a negative INTEGER value.

  This statement returns 12 because the second timestamp is 12 days greater than the first one. Both timestamps have a default time of 00:00:00.

  SQL

  ```
  SELECT {fn TIMESTAMPDIFF(SQL_TSI_DAY,'2022-4-1','2022-4-13')}
  ```

  Example: Calculate Difference Between Timestamps

Arguments

interval

The type of time or date interval that the returned timestamp difference represents, specified as one of these timestamp intervals:

- `SQL_TSI_FRAC_SECOND` — Fractional second intervals
- `SQL_TSI_SECOND` — Second intervals
- `SQL_TSI_MINUTE` — Minute intervals
- `SQL_TSI_HOUR` — Hour intervals
- `SQL_TSI_DAY` — Day intervals
- `SQL_TSI_WEEK` — Week intervals
- `SQL_TSI_MONTH` — Month intervals
- `SQL_TSI_YEAR` — Year intervals

startDate,endDate

Timestamp value expressions representing the start and end date being compared, specified as one of these values:

- %Library.TimeStamp data type format (yyyy-mm-dd hh:mm:ss.ffff)
- %Library.PosixTime data type format (an encoded 64-bit signed integer)

You can specify these timestamp intervals with or without enclosing quotation marks, using single quotes or double quotes. They are not case-sensitive.

If either *startDate* or *endDate* uses the %Library.TimeStamp format, these rules apply:

- If either timestamp expression specifies only a time value and *interval* specifies a date interval (days, weeks, months, or years), the missing date portion of the timestamp defaults to '1900–01–01' before calculating the resulting interval count.

- If either timestamp expression specifies only a date value and *interval* specifies a time interval (hours, minutes, seconds, fractional seconds), the missing time portion of the timestamp defaults to '00:00:00.000' before calculating the resulting interval count.

- You can include or omit fractional seconds of any number of digits of precision. SQL_TSI_FRAC_SECOND returns a difference of fractional seconds as an integer count of thousandths of a second (three digits of precision). %Library.PosixTime values always includes six digits of precision.

Examples

Calculate Difference Between Timestamps

This statement returns 7 because the second timestamp (2021-12-20 12:00:00) is 7 months greater than the first one:

SQL

```
SELECT {fn TIMESTAMPDIFF(SQL_TSI_MONTH,
    '2021-5-19 00:00:00','2021-12-20 12:00:00')}
```

This statement returns 566 because the second timestamp ('12:00:00') is 566 minutes greater than the first one (02:34:12):

SQL

```
SELECT {fn TIMESTAMPDIFF(SQL_TSI_MINUTE,'02:34:12','12:00:00')}
```

This statement returns -1440 because the second timestamp is one day (1440 minutes) less than the first one:

SQL

```
SELECT {fn TIMESTAMPDIFF(SQL_TSI_MINUTE,'2021-12-06','2021-12-05')}
```

Limitations

- You can use **TIMESTAMPDIFF** only as an ODBC scalar function, which requires the curly brace syntax. To perform similar time and date comparison operations on a timestamp, use the DATEDIFF function.

More About

Range and Value Checking

Prior to performing the difference calculation, **TIMESTAMPDIFF** performs these checks on input values:

- All specified parts of *startDate* and *endDate* are valid. Time values can be wholly or partially omitted. If *startDate* or *endDate* specifies an incomplete time, **TIMESTAMPDIFF** supplies zeros for the unspecified parts.

- Date strings are complete and properly formatted with the appropriate number of elements, number of digits for each element, and the appropriate separator character. Years must be specified as four digits. An invalid date value results in an SQLCODE -8 error.

- Date values are within a valid range. Years: 0001 through 9999. Months: 1 through 12. Days: 1 through 31. Hours: 00 through 23. Minutes: 0 through 59. Seconds: 0 through 59. The number of days in a month must match the month and year. For example, the date '02–29' is valid only if the specified year is a leap year. An invalid date value results in an SQLCODE -8 error.

- Date values contain only canonical integer values. Exception: Months and days with values less than 10 (month and day) can include a leading zero. Therefore, a day value of '07' or '7' is valid, but '007', '7.0' or '7a' are not valid.

- Hour values less than 10 include a leading zero. Omitting this leading zero results in an SQLCODE -8 error.

See Also

- TIMESTAMPADD
- DATEDIFF
- TO_POSIXTIME
- TO_TIMESTAMP

TO_CHAR (SQL)

A string function that converts a date, timestamp, or number to a formatted character string.

Synopsis

```
TO_CHAR(expression,format)

TO_CHAR(expression)

TOCHAR(...)
```

Description

- **TO_CHAR(***expression,format***)** converts a date, time, timestamp (date and time), or number expression to a character string according to the specified format string.

  This statement converts the current date to the format `'MONTH DD, YYYY'`, where `MONTH` is the full month name, `DD` is the two-digit day of the month, and `YYYY` is the four-digit year.

  ### SQL

  ```
  SELECT TO_CHAR(CURRENT_DATE,'MONTH DD, YYYY')
  ```

  Examples:

  – Convert Dates to Formatted Date Strings

  – Convert Times to Formatted Time Strings

  – Convert Timestamps to Formatted Date and Time Strings

  – Convert Numbers to Formatted Numeric Strings

  TO_CHAR(*expression***)** converts a date, time, timestamp, or number expression according to the default Logical mode format for the expression type.

  – *Date expressions* and *time expressions* convert to the InterSystems SQL Logical $HOROLOG format, which is a string of two comma-separated integers that represent a date and time. The first integer is the number of days since December 31, 1840. The second integer is the number of seconds since midnight of the current day.

  – *Timestamp expressions* convert to the format `YYYY-MM-DD HH:MI:SS`.

  – *Number expressions* convert to integers. Any leading zeros or plus signs are removed, and the number is truncated at the first nonnumeric character, such as a comma or period.

  This statement converts the current date and time, represented as a timestamp, to a character string of the format `YYYY-MM-DD HH:MI:SS`.

  ### SQL

  ```
  SELECT TO_CHAR(CURRENT_TIMESTAMP)
  ```

- **TOCHAR(...)** is equivalent to **TO_CHAR(...)**.

Arguments

expression

A logical date, time, timestamp, or number expression to be converted to a character string according to the format specified by *format*. If *expression* is null, **TO_CHAR** returns null.

Date Expressions

To convert date expressions, *expression* must be an integer or string in $HOROLOG format.

If *expression* is an invalid date, (for example, February 30), InterSystems IRIS® issues an SQLCODE -400 error.

If *expression* represents a date before 12/31/1840, then to convert the date, you must use the Julian date format (*format* argument = 'J'). For more details, see Julian Date Conversion.

Time Expressions

To convert time expressions, *expression* must be in one of these formats:

- A **$HOROLOG** time integer (the time component of **$HOROLOG**), where *expression* is a valid Logical time integer in the range 0 through 86399. Do not supply a full **$HOROLOG** value with both date and time components (such as 64701,42152). **TO_CHAR** time conversion converts only the first component of **$HOROLOG**, the date component, to a formatted time string and ignores the second component, the time component.

- A Logical timestamp value. The value for *expression* must be of the %TimeStamp data type (not a string data type) in the format YYYY-MM-DD hh:mm:ss. If you specify only a time format in *format*, then **TO_CHAR** ignores the date component of the timestamp converts only the time component. For example, SYSDATE is a Logical timestamp.

- A time value in standard ODBC time format. The value for *expression* must be in the format hh:mm:ss and can be a string.

- A time value in local time format using the current NLS locale settings. For example, if the NLS TimeSeparator is set to "^", the value for *expression* can be in the format hh^mm^ss and can be a string.

If *expression* is an invalid time, (for example, 6:61 P.M.), InterSystems IRIS issues an SQLCODE -400 error.

Timestamp Expressions

To convert timestamp expressions, *expression* must be of the format YYYY-MM-DD HH:MI:SS, or one of the following valid variants:

- For month and date values less than 10, leading zeros are optional. If the leading zero is omitted, it is also omitted in the returned date.

- The seconds value can be omitted, but you must specify the colon indicating its place (for example, HH:MI:). In the returned time, the seconds default to 00.

- The seconds value can include fractional seconds (for example, HH:MI:SS.fff). In the returned time, these fractional seconds are truncated.

- A timestamp must include a time portion, even if *format* does not specify time formatting.

If *expression* is not a valid timestamp format, **TO_CHAR** interprets it as an integer, ending interpretation when it encounters the first non-integer character.

If *format* is a date or timestamp format, **TO_CHAR** interprets *expression* as a **$HOROLOG** date integer. Thus 2010-03-23 12-15:23 (note erroneous hyphen in time value) is interpreted as the **$HOROLOG** date 2010 (1846-07-03 12:00:00 AM).

If *expression* is an invalid date or time, (for example, February 30 or 6:61 P.M.), InterSystems IRIS issues an SQLCODE -400 error.

Number Expressions

To convert number expressions, *expression* must be a numeric data type or a numeric string. **TO_CHAR** truncates strings at the first nonnumeric integer. It interprets a string with no leading numeric values as 0.

format

A character code that specifies a date, timestamp, or number format for the *expression* conversion.

- If you specify *format* with an invalid date, time, or timestamp code element (for example, YYYYY, MIN, HH48), **TO_CHAR** returns the format code literal for the invalid code element. For all other valid code elements, it returns the date, time, or timestamp conversion values.

- If **TO_CHAR** cannot recognize any *format* code elements (for example, *format* is an empty string) or if a number format has fewer digits than the *expression* value, **TO_CHAR** returns pound signs (#) in place of the original characters. This is true when *expression* begins with at least two integer digits. Otherwise, **TO_CHAR** returns NULL.

These tables list the valid format codes that you can specify for each expression type: date, time, date and time (timestamp), and number.

Table G–7: Date Formats

Format Code	Meaning
D	Day of week (1–7). By default, 1 is Sunday (the first day of the week), but you can configure this value. For more details, see DAYOFWEEK.
DD	Two-digit day of month (01–31).
DY	Abbreviated name of day, as specified by the WeekdayAbbr property in the current locale.
	Defaults: Sun Mon Tue Wed Thu Fri Sat
DAY	Name of day, as specified by the WeekdayName property in the current locale.
	Defaults: Sunday Monday Tuesday Wednesday Thursday Friday Saturday
MM	Two-digit month number (01–12; 01 = JAN).
MON	Abbreviated name of month, as specified by the MonthAbbr property in the current locale.
	Defaults (case-insensitive): Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
MONTH	Full name of the month, as specified by the MonthName property in the current locale.
	Defaults (case-insenitive) January February March April May June July August September October November December
YYYY	Four-digit year.
YYY	Last 3 digits of the year.
YY	Last 2 digits of the year.
Y	Last digit of the year.
RRRR	Four-digit year.

Format Code	Meaning
RR	Last 2 digits of the year.
DDD	Day of the year (number of days since January 1 of specified year)
J	Julian date (number of days since January 1, 4712 BCE). For more details, see Julian Date Conversion.

Separator characters are required between the date format elements, with the exception of the following format strings: YYYYMMDD, DDMMYYYY, and YYYYMM. The last of these returns the year and month values and ignores the day of the month.

Locales mentioned in the *format* code definitions refer to the same locales described in the ObjectScript **$ZDATE** and **$ZDATEH** documentation.

Table G–8: Time Formats

Format Code	Meaning
HH	Hour of day (1–12)
HH12	Hour of day (1–12)
HH24	Hour of day (0–23)
MI	Minute (0– 59)
SS	Second (0–59)
SSSSS	Seconds since midnight (0–86388)
AM / PM	Meridian Indicator (AM = before noon, PM = after noon). Converts a time value to 12-hour format with the appropriate AM or PM suffix. The returned AM or PM suffix is derived from the time value, not from the format code you specified. In *format*, you can use either AM or PM. They are functionally identical.

When converting times to strings, *format* must be a string that contains only the time format codes shown in the table. If format includes any other codes, then **TO_CHAR** interprets the *expression* as a date instead.

When converting timestamps to formatted datetime strings, format must be a string containing the date and time format codes shown in the "Date Formats" and "Time Formats" tables. To perform this conversion, *expression* must be a valid Logical timestamp value.

Table G–9: Number Formats

Format Code	Description	Example
9	Return value with the specified number of digits. • Positive values include a leading space. • Negative values include a minus sign. • Leading zeros are blank, except for a zero value, which returns a zero for the integer part of the fixed-point number.	9999
0	Return leading or trailing zeros.	09999 99990

Format Code	Description	Example
$	Return value with a leading dollar sign. For positive numbers, the dollar sign is preceded by a blank space.	$9999
B	Return blanks for the integer part of a fixed-point number when the integer part is zero (regardless of 0 in the *format* argument).	B9999
S	Return value with a leading or trailing plus sign "+" if positive and a leading or trailing minus sign "-" if negative.	S9999 9999S
D	Return a decimal separator character in the specified position. The DecimalSeparator used is the one defined for the locale. The default is a period ".". Only one "D" is allowed in the *format* argument.	99D99
G	Return a numeric group separator character in the specified positions. The NumericGroupSeparator used is the one defined for the locale. The default is a comma ",". No numeric group separators can appear to the right of the decimal separator.	9G999
FM	Return a value with no leading or trailing blanks.	FM90.9
,	Return a comma in the specified position. No comma can appear to the right of the decimal. The *format* argument cannot begin with a comma.	9,999
.	Return a decimal point (that is, a period ".") in the specified position. Only one "." is allowed in the *format* argument.	99.99

format can specify the decimal separator and the numeric group separator either as a literal character, or as the current value of the locale's `DecimalSeparator` and `NumericGroupSeparator`. You can determine the current locale values using ObjectScript as follows:

ObjectScript

```
write ##class(%SYS.NLS.Format).GetFormatItem("DecimalSeparator"),!
write ##class(%SYS.NLS.Format).GetFormatItem("NumericGroupSeparator")
```

If *format* contains fewer integer digits than the input numeric expression, **TO_CHAR** does not return a number. Instead, it returns a string of two or more pound signs (##). The number of pound signs represents the length of the current *format* argument, plus one.

If *format* contains fewer decimal digits than the input numeric expression, **TO_CHAR** rounds the number to the specified number of decimal digits. If no decimal format is provided, **TO_CHAR** rounds the number to an integer.

Examples

Convert Dates to Formatted Date Strings

This statement uses **TO_CHAR** to convert **$HOROLOG** date integers or full **$HOROLOG** string values to formatted date strings or date and time strings:

SQL

```
SELECT
  TO_CHAR(66256,'YYYY-MM-DD') AS Date2FormattedDate,
  TO_CHAR(66256,'YYYY-MM-DD HH24:MI:SS') AS Date2FormattedDateTime,
  TO_CHAR('66256,50278','YYYY-MM-DD') AS DateTime2FormattedDate,
  TO_CHAR('66256,50278','YYYY-MM-DD HH24:MI:SS') AS DateTime2FormattedDateTime
```

In this statement, each **TO_CHAR** call takes a date integer and returns a date string formatted according to the *format* string argument:

SQL

```
SELECT
  TO_CHAR(66256,'MM/DD/YYYY'),          /* returns 02/22/2018 */
  TO_CHAR(66256,'DAY MONTH DD, YYYY')   /* returns Thursday February 22, 2018 */
```

This statement converts a date integer to a formatted date string. Invalid *format* characters are passed through to the output string as literals. It returns the string `The date 05/27/2022 should be noted`.

SQL

```
SELECT TO_CHAR(66256,'The date MM/DD/YYYY should be noted')
```

This statement converts date expressions to the day of the year, defined as the number of days elapsed since January 1 of the specified year. To use this syntax, the date expression must be in **$HOROLOG** format. The time value in the second **TO_CHAR** call is ignored. In the two **TO_CHAR** calls, the day-of-year format element (DDD) and the year elements (YYYY and YY) appear in a different order.

SQL

```
SELECT
  TO_CHAR('66235','DDD days into YYYY'),
  TO_CHAR('66235,12345','Year YY: DDD days elapsed')
```

In this statement, **TO_CHAR** returns an incorrect date value, because it interprets the separators as minus signs. Therefore, it evaluates the expression as 2022 – 5 – 2 = 2015, which in **$HOROLOG** integer format corresponds to the date 1846-07–08.

SQL

```
SELECT TO_CHAR(2022-05-02,'YYYY-MM-DD') -- Incorrect usage
```

Convert Times to Formatted Time Strings

This statement causes `'66256'` to be interpreted as the time value `06:24:16 PM`.

SQL

```
SELECT TO_CHAR('66256','HH12:MI:SS PM')
```

This statement converts the time portions of two Logical timestamps to formatted time strings. Because *format* does not support fractional seconds, the fractional seconds in *expression* are truncated.

SQL

```
SELECT TO_CHAR(SYSDATE,'HH12:MI:SS PM'),
       TO_CHAR(CURRENT_TIMESTAMP(6),'HH12:MI:SS PM')
```

Convert Timestamps to Formatted Date and Time Strings

This statement returns the current system date (a timestamp), and the current system date converted for display with two different formats:

SQL

```
SELECT
  SYSDATE,
  TO_CHAR(SYSDATE,'MM/DD/YYYY HH:MI:SS'),
  TO_CHAR(SYSDATE,'DD MONTH YYYY at SSSSS seconds')
```

Any characters used in the *format* string that are not format codes are returned in place in the resulting string.

Convert Numbers to Formatted Numeric Strings

This statement converts the number 1000 to strings with varying numbers of digit format codes.

- In the first conversion, the number has more digits than specified digit format codes. **TO_CHAR** returns pound symbols equal to the number of digits in the number.

- In the second conversion, the number has the same number of digits as specified digit format codes. **TO_CHAR** returns the number in character string form. Because the number is an unsigned positive integer, **TO_CHAR** prepends a leading zero to the numeric string.

- In the third conversion, the number has fewer digits than specified digit format codes. **TO_CHAR** returns the number in character string form and prepends two leading zeros: one because the number is an unsigned positive integer one for the extra digit format code.

SQL

```
SELECT
  TO_CHAR(1000,'999'),   -- '####'
  TO_CHAR(1000,'9999'),  -- ' 1000'
  TO_CHAR(1000,'99999')  -- '  1000'
```

The statement shows the use of separator characters:

- The first conversion returns the string: ' 1,000.00'.

- The second conversion might return the same value, but the separator characters displayed depend on the locale setting.

SQL

```
SELECT
  TO_CHAR(1000,'9,999.99'),
  TO_CHAR(1000,'9G999D99')
```

This statement shows the use of positive and negative signs. The leading space appears only before a positive number with no sign formatting. No leading space appears before a negative number or any signed number, regardless of the placement of the sign.

SQL

```
SELECT
  TO_CHAR(10,'99.99'),    -- ' 10.00'
  TO_CHAR(-10,'99.99'),   -- '-10.00'
  TO_CHAR(10,'S99.99'),   -- '+10.00'
  TO_CHAR(-10,'S99.99'),  -- '-10.00'
  TO_CHAR(10,'99.99S'),   -- '10.00+'
  TO_CHAR(-10,'99.99S')   -- '10.00-'
```

This statement shows the use of the "FM" format to override the default leading blank for unsigned positive numbers:

SQL

```
SELECT
  TO_CHAR(12345678.90,'99,999,999.99'),  -- ' 12,345,678,90'
  TO_CHAR(12345678.90,'FM99,999,999.99') -- '12,345,678,90'
```

This statement shows the use of the leading dollar sign. The dollar sign is always preceded either by a sign or by a blank character.

SQL

```
SELECT
  TO_CHAR(1234567890,'$9G999G999G999'),   -- ' $1,234,567,890'
  TO_CHAR(1234567890,'S$9G999G999G999'),  -- '+$1,234,567,890'
  TO_CHAR(12345678.90,'$99G999G999D99')   -- ' $1,234,567,8.90'
```

The statement shows what happens when the *format* argument contains fewer decimal (fractional) digits than the input numeric expression. The returned numbers are rounded to 1234567.5 and 1234568, respectively.

SQL

```
SELECT
  TO_CHAR(1234567.4999,'9999999.9'),
  TO_CHAR(1234567.91,'9999999')
```

More About

Julian Date Conversion

The Julian date format enables you to convert dates before December 31, 1840 to character strings. To use this format, specify the *format* argument of **TO_CHAR** as 'J' or 'j'. Using this format, you can convert a date value for data type %Date or %TimeStamp to a seven-digit Julian date integer, with leading zeros added when necessary. For example:

SQL

```
SELECT
  TO_CHAR('1776-07-04','J') AS UnitedStatesStart, --2369916
  TO_CHAR('-0031-09-02','J') AS RomanEmpireStart  --1709980
```

The returned integer is a count of days from January 1, 4712 BCE. The maximum *expression* value that you can convert to a Julian date is '9999-12-31' (Julian day count 5373484). The minimum value is '-4712-01-01' (Julian day count 0000001).

By default, the %Date data type does not represent dates prior to December 31, 1840. However, you can redefine the MINVAL parameter for this data type to permit representation of earlier dates as negative integers, with the limit of January 1, Year 1. This representation of dates as negative integers is not compatible with the Julian date format described here. For more details, see Data Types.

The Julian day count value of 1721424 returns January 1st of Year 1 (1-01-01) in the Julian calendar. Julian day counts less than this values return BCE dates, which are displayed with the year preceded by a minus sign.

TO_CHAR permits you to return a Julian day count corresponding to a date expression. **TO_DATE** permits you to return a date expression corresponding to a Julian day count, as shown in this example:

SQL

```
SELECT
  TO_CHAR('1776-07-04','J') AS JulianCount, -- 2369916
  TO_DATE(2369916,'J') AS JulianDate        -- 1776-07-04
```

Related SQL Functions

- **TO_CHAR** converts a date integer, timestamp, or a number to a string.

- TO_DATE performs the reverse operation for dates. It converts a formatted date string to a date integer.

- TO_TIMESTAMP performs the reverse operation for timestamps. It converts a formatted date and time string to a standard timestamp.

- TO_NUMBER performs the reverse operation for numbers. It converts a numeric string to a number.

- CAST and CONVERT perform DATE, TIMESTAMP, and NUMBER data type conversions.

Alternatives

To perform equivalent conversions in ObjectScript, use the **TOCHAR()** method:

```
$SYSTEM.SQL.Functions.TOCHAR(expression,format)
```

See Also

- SQL functions: CONVERT, TO_DATE, TO_NUMBER

- ObjectScript functions: $FNUMBER, $ZDATE

TO_DATE (SQL)

A date function that converts a formatted string to a date.

Synopsis

```
TO_DATE(dateString) TO_DATE(dateString,format)  TODATE(...)
```

Description

The **TO_DATE** function converts date strings in various formats to a date integer value of data type DATE. This function is used to input dates in various string formats and store them using a standard InterSystems IRIS® representation.

TO_DATE returns an integer count of the number of days since December 31, 1840, where 0 represents December 31, 1840, the minimum value of -672045 represents January 1, 0001, and the maximum value of 2980013 represents December 31, 9999.

- **TO_DATE(*dateString*)** parses the date string using the default format of DD MON YYYY.

  This statement converts the date string 22 Feb 2022 to the integer 66162.

  SQL

  ```
  SELECT TO_DATE('22 FEB 2022')
  ```

  Example: Default Date Format

- **TO_DATE(*dateString*,*format*)** parses the date string using the specified format string. The date elements of *dateString* must correspond to the format elements of *format*.

  This statement performs the same conversion as in the previous syntax, but it enables you to specify the date string using a custom format.

  SQL

  ```
  SELECT TO_DATE('2-22-22','M-DD-YY')
  ```

  Examples:

  - Specified Date Format
  - Standalone Date Element Formats
  - Day of the Year Conversion (DDD Format)
  - Two-Digit Year Conversion (RR and RRRR Formats)
  - Set Default Date Column Values

- **TODATE(...)** is equivalent to **TO_DATE(...)**.

Arguments

dateString

The *dateString* argument is a string expression that specifies the date string to be converted to a date. The underlying data type of *dateString* must be CHAR or VARCHAR2.

Each character of *dateString* must correspond to the *format* string, with these exceptions:

- Leading zeros can be included or omitted, with the exception of a *dateString* without separator characters.

- Years can be specified with two digits or four digits.

- Month names can be specified in full or as the first three letters of the name. Only the first three letters must be correct. Month names are not case-sensitive.

- Time values appended to a date are ignored.

The earliest date you can specify is December 31, 1840, which InterSystems IRIS represents as logical integer 0. To specify earlier dates, use the Julian data format. See Julian Dates (J Format).

format

The *format* argument is a date string that specifies the format of the *dateString*. **TO_DATE** converts the date elements of *dateString* according to the format elements in the corresponding positions of *format*. The elements of *format* follow these rules:

- Format elements are not case-sensitive.

- Almost any sequence or number of format elements is permitted.

- Format strings separate their elements with non-alphanumeric separator characters (for example, a space, slash, or hyphen) that match the separator characters in the *dateString*. This use of specified date separator characters does not depend on the DateSeparator defined for your NLS (National Language Support) locale.

- The following date format strings do not require separator characters: MMDDYYYY, DDMMYYYY, YYYYMMDD, and YYYYDDMM. The incomplete date format YYYYMM is also supported and assumes a DD value of 01. In these formats, the MM and DD values require leading zeros.

This table lists the valid date format elements. The uppercase form is shown, but these elements are not case-sensitive.

Element	Meaning
DD	Two-digit day of month (01-31). Leading zeros are not required, unless *format* contains no date separator characters.
MM	Two-digit month number (01-12; 01 = January). Leading zeros are not required, unless *format* contains no date separator characters. In Japanese and Chinese, a month number consists of a numeric value followed by the ideogram for "month".
MON	Abbreviated name of month, as specified by the MonthAbbr property in the current locale. By default, in English, this is the first three letters of the month name. In other locales, month abbreviations can be more than three letters long and might not consist of the first letters of the month name. A period character is not permitted. Not case-sensitive.
MONTH	Full name of the month, as specified by the MonthName property in the current locale. Not case-sensitive.
YYYY	Four-digit year.
YY	Last two digits of the year. The first two digits of a two-digit year default to 19.
RR / RRRR	Two-digit year to four-digit year conversion. For more details, see Two-Digit Year Conversion (RR and RRRR Formats).
DDD	Day of the year. The count of days since January 1. See Day of the Year Conversion (DDD Format).

Element	Meaning
J	Julian date. Used to represent dates prior to December 31, 1840. See Julian Dates (J format).

A **TO_DATE** *format* can also include a D (day of week number), DY (day of week abbreviation), or DAY (day of week name) element. However, these format elements are not validated or used to determine the return value. For more details on these *format* elements, see TO_CHAR.

If you omit *format*, **TO_DATE** parses the date string using the default format of DD MON YYYY. For example, '22 Feb 2018'. To change the default date format system-wide, you can modify the TODATEDefaultFormat configuration parameter.

To view the current setting, call **$SYSTEM.SQL.CurrentSettings()**, which displays the TO_DATE() Default Format setting.

Examples

Default Date Format

This statement specifies date strings that are parsed using the default date format. Both of these are converted to the DATE data type internal value of 60537:

SQL

```
SELECT
  TO_DATE('29 September 2018'),
  TO_DATE('29 SEP 2018')
```

This statement specifies date strings with two-digit years with *format* default. Note that two-digit years default to 1900 through 1999. Thus, the internal DATE value is 24012:

SQL

```
SELECT
  TO_DATE('29 September 06'),
  TO_DATE('29 SEP 06')
```

Specified Date Format

This statement specifies date strings in various formats. All of these are converted to the DATE data type internal value of 64701.

SQL

```
SELECT
  TO_DATE('2018 Feb 22','YYYY MON DD'),
  TO_DATE('FEBRUARY 22, 2018','month dd, YYYY'),
  TO_DATE('2018***02***22','YYYY***MM***DD'),
  TO_DATE('02/22/2018','MM/DD/YYYY')
```

This statement specifies date formats that do not require element separators. They return the date internal value of 64701.

SQL

```
SELECT
  TO_DATE('02222018','MMDDYYYY'),
  TO_DATE('22022018','DDMMYYYY'),
  TO_DATE('20182202','YYYYDDMM'),
  TO_DATE('20180222','YYYYMMDD')
```

This statement specifies the YYYYMM date format. It does not require element separators. It supplies 01 for the missing day element, returning the date 64800 (June 1, 2018):

SQL

```
SELECT TO_DATE('201806','YYYYMM')
```

Standalone Date Element Formats

In the *format* argument, you can specify DD, DDD, MM, or YYYY as standalone date strings. Because these format strings omit the month, year, or both the month and year, InterSystems IRIS interprets them as referring to the current month and year.

DD returns the date for the specified day in the current month of the current year. For example:

SQL

```
SELECT TO_DATE('24','DD')
```

DDD returns the date for the specified number of days elapsed in the current year. For example:

SQL

```
SELECT TO_DATE('300','DDD')
```

MM returns the date for the first day of the specified month in the current year. For example:

SQL

```
SELECT TO_DATE('8','MM')
```

YYYY returns the date for the first day of the current month of the specified year. For example:

SQL

```
SELECT TO_DATE('2022','YYYY')
```

Two-Digit Year Conversion (RR and RRRR Formats)

The YY format converts a two-digit year value to four digits by appending 19. For example, 07 becomes 1907 and 93 becomes 1993. The RR and RRRR formats provide more flexible two-digit to four-digit year conversion.

The RR format conversion is based on the current year.

- If the current year is in the first half of a century:

    - Two-digit years from 00 through 49 are expanded to a four-digit year in the current century.

    - Two-digit years from 50 through 99 are expanded to a four-digit year in the previous century.

    This statement shows the display format of dates that **TO_DATE** returns when the current year is between 2000 and 2050.

    SQL

    ```
    SELECT
      TO_DATE('29 September 00','DD MONTH RR'), -- 09/29/2000
      TO_DATE('29 September 18','DD MONTH RR'), -- 09/29/2018
      TO_DATE('29 September 49','DD MONTH RR'), -- 09/29/2049
      TO_DATE('29 September 50','DD MONTH RR'), -- 09/29/1950
      TO_DATE('29 September 77','DD MONTH RR')  -- 09/29/1977
    ```

- If the current year is in the second half of a century, all two-digit years are expanded to a four-digit year in the current century.

This statement shows the display format of dates that **TO_DATE** returns when the current year is between 2050 and 2099.

SQL

```
SELECT
  TO_DATE('29 September 00','DD MONTH RR'), -- 09/29/2000
  TO_DATE('29 September 21','DD MONTH RR'), -- 09/29/2021
  TO_DATE('29 September 49','DD MONTH RR'), -- 09/29/2049
  TO_DATE('29 September 50','DD MONTH RR'), -- 09/29/2050
  TO_DATE('29 September 77','DD MONTH RR')  -- 09/29/2077
```

Using the RRRR format, you can input a mix of two-digit and four-digit years. **TO_DATE** passes four-digit years through unchanged. **TO_DATE** converts two-digit years to four-digit years by using the RR format algorithm described earlier in this example.

This statement shows the display format of dates that **TO_DATE** returns when the current year is between 2000 and 2050.

SQL

```
SELECT
  TO_DATE('29 September 2021','DD MONTH RRRR'), -- 09/29/2021
  TO_DATE('29 September 21','DD MONTH RRRR'),   -- 09/29/2021
  TO_DATE('29 September 1949','DD MONTH RRRR'), -- 09/29/1949
  TO_DATE('29 September 49','DD MONTH RRRR'),   -- 09/29/2049
  TO_DATE('29 September 1950','DD MONTH RRRR'), -- 09/29/1950
  TO_DATE('29 September 50','DD MONTH RRRR')    -- 09/29/1950
```

Day of the Year Conversion (DDD Format)

You can use the DDD format to convert the day of the year (that is, the number of days elapsed since January 1) to an actual date. To perform this conversion:

- The *format* argument must contain the DDD format element and optionally a year format such as YYYY, YY, RR, or RRRR. You can specify these elements in any order but they must include a separator character between them. If you omit the year element, then **TO_DATE** defaults to the current year.

- The *dateString* argument must contain corresponding *day* and *year* values, where:

  – *day* is an integer in the range 1 through 365 (366 if *year* is a leap year).

  – *year* is a year within the standard InterSystems IRIS date range: 1841 through 9999.

This statement returns the 60th day of the year 2022.

SQL

```
SELECT TO_DATE('2022:60','YYYY:DDD') -- 03/01/2022
```

TO_DATE passes month elements through unchanged. If a format string contains both a DD and a DDD element, **TO_DATE** processes the DDD element and ignores the DD element. For example, this statement returns 2/29/2020 (the 60th day of 2020), not 12/31/2020:

SQL

```
SELECT TO_DATE('2020-12-31-60','YYYY-MM-DD-DDD')
```

TO_DATE returns a date expression containing the day of the year, not the day of the year itself. To return this day value, use **TO_CHAR**.

Set Default Date Column Values

When creating a table using the CREATE TABLE command, you can use the **TO_DATE** function to set the default value of a column. For example:

SQL

```
CREATE TABLE MyTable
(ID NUMBER(12,0) NOT NULL,
End_Year DATE DEFAULT TO_DATE('12-31-2021','MM-DD-YYYY') NOT NULL)
```

More About

Julian Dates (J Format)

The Julian date format enables you to represent dates before December 31, 1840. To use this format, specify the format argument of **TO_DATE** as `'J'` or `'j'`. Using this format, you can convert a seven-digit internal numeric value (a Julian day count) to a formatted date. For example, this statement returns `1585-01-31` in Logical or ODBC format and `01/31/1585` in Display format.

SQL

```
SELECT TO_DATE(2300000,'J')
```

The Julian day count value of 1721424 returns January 1st of Year 1 (`1-01-01`) in the Julian calendar. Julian day counts less than this values return BCE dates, which are displayed with the year preceded by a minus sign.

By default, the %Date data type does not represent dates prior to December 31, 1840. However, you can redefine the MINVAL parameter for this data type to permit representation of earlier dates as negative integers, with the limit of January 1, Year 1. This representation of dates as negative integers is not compatible with the Julian date format described here. For more details, see Data Types.

A Julian day count is always represented internally as a seven-digit number, with leading zeros when necessary. **TO_DATE** allows you to input a Julian day count without the leading zeros. The highest permitted Julian date is 5373484, which returns `12/31/9999`. The lowest permitted Julian date is 0000001, which returns `01/01/-4712` (01/01/4713 BCE). Any value outside this range generates an SQLCODE -400 error.

Julian day counts prior to 1721424 (1/1/1) are compatible with other software implementations, such as Oracle. They are *not* identical to BCE dates in ordinary usage. In ordinary usage, there is no Year 0 and dates go from 12/31/-1 to 1/1/1. In Oracle usage, the Julian dates 1721058 through 1721423 are simply invalid, and return an error. In InterSystems IRIS, these Julian dates return the non-existent Year 0 as a place holder. Thus calculations involving BCE dates must be adjusted by one year to correspond to common usage. This should not affect the conversion of dates and Julian day counts using **TO_CHAR** and **TO_DATE**, but it might affect some calculations made using Julian day counts. Also, be aware that these date counts do not take into account changes in date caused by the Gregorian calendar reform.

TO_DATE permits you to return a date expression corresponding to a Julian day count. **TO_CHAR** permits you to return a Julian day count corresponding to a date expression, as shown in this example:

SQL

```
SELECT
  TO_CHAR('1776-07-04','J') AS JulianCount, -- 2369916
  TO_DATE(2369916,'J') AS JulianDate        -- 1776-07-04
```

Related SQL Functions

- **TO_DATE** converts a formatted date string to a date integer.

- TO_CHAR performs the reverse operation. It converts a date integer to a formatted date string.

- [TO_TIMESTAMP](#) converts a formatted date and time string to a standard timestamp.

- [CAST](#) and [CONVERT](#) perform DATE data type conversion.

Alternatives

To perform equivalent date conversions in ObjectScript, use the **TODATE()** method:

```
$SYSTEM.SQL.Functions.TODATE(dateString,format)
```

See Also

- SQL functions: [CAST](#), [CONVERT](#), [TO_CHAR](#), [TO_TIMESTAMP](#)

- ObjectScript functions: [$ZDATE](#), [$ZDATEH](#)

TO_NUMBER (SQL)

A string function that converts a string expression to a value of NUMERIC data type.

Synopsis

```
TO_NUMBER(stringExpression)

TONUMBER(stringExpression)
```

Description

- **TO_NUMBER(*stringExpression*)** converts the input string expression to a canonical number of data type NUMERIC. If the string expression is of data type DOUBLE, **TO_NUMBER** returns a number of data type DOUBLE. All other types that do not appear in the following table return the type of *stringExpression*:

Type of stringExpression	Type returned
VARCHAR, VARBINARY, TIME	NUMERIC
BIT	TINYINT
DATE	INTEGER
TIMESTAMP POSIXTIME	BIGINT

This query returns the addresses of a specific street in ascending numerical order. If you do not convert the street addresses by specifying `ORDER BY Home_Street` and do not convert to numbers, then the addresses follow string collation order (1, 10, 100, 2, 20, 200, and so on).

SQL

```
SELECT Name,Home_Street FROM Sample.Person WHERE Home_Street LIKE '%Oakhurst%'
ORDER BY TO_NUMBER(Home_Street)
```

Examples:

- String-to-Number Conversion Operations

- Format Modes of Converted Strings

- **TONUMBER(*stringExpression*)** is equivalent to **TO_NUMBER(*stringExpression*)**.

Arguments

stringExpression

The string expression to be converted. The expression can be the name of a column, a string literal, or the result of another function that has an underlying data type of CHAR or VARCHAR2.

Examples

String-to-Number Conversion Operations

This example shows the different operations that **TO_NUMBER** performs to convert numeric strings into canonical numbers. The returned results shown in the SQL comments are in Logical mode. For more details on how converted numbers are displayed, see Format Modes of Converted Strings.

TO_NUMBER resolves leading plus and minus signs.

SQL

```
SELECT TO_NUMBER('-+123 feet') -- -123
```

SQL

```
SELECT TO_NUMBER('+-+-123 feet') -- 123
```

TO_NUMBER also expands exponential notation ("E" or "e").

SQL

```
SELECT TO_NUMBER('1e3') -- 1000
```

SQL

```
SELECT TO_NUMBER('1E-3') -- .001
```

TO_NUMBER halts conversion when it encounters a nonnumeric character, such as a letter or a numeric group separator.

SQL

```
SELECT TO_NUMBER('7dwarves') -- 7
```

If the first character of the string expression is not numeric, or if the expression is an empty string (") or -0, **TO_NUMBER** returns 0.

SQL

```
SELECT TO_NUMBER('question3') -- 0
```

SQL

```
SELECT TO_NUMBER('') -- 0
```

SQL

```
SELECT TO_NUMBER('-0') -- 0
```

TO_NUMBER does not resolve arithmetic operations. For example, in this string, **TO_NUMBER** halts conversion at the "+" character and returns 2.

SQL

```
SELECT TO_NUMBER('2+4') -- 2
```

If NULL is specified for the string expression, **TO_NUMBER** returns null.

Format Modes of Converted Strings

The number format of the returned query results can differ depending on whether you use Logical mode, ODBC mode, or Display mode.

Unless the string expression is a DOUBLE, the **TO_NUMBER** function returns a number of type NUMERIC. The NUMERIC data type has a default scale of 2. Therefore, when running queries in Display mode, InterSystems SQL displays the returned results with 2 decimal places.

SQL

```
SELECT TO_NUMBER('-15 degrees F') -- Display Mode: -15.00
```

Additional fractional digits are rounded to two decimal places.

SQL

```
SELECT TO_NUMBER('-15.835 degrees F') -- Display Mode: -15.84
```

Trailing zeros are also resolved to two decimal places.

SQL

```
SELECT TO_NUMBER('-15.60000 degrees F') -- Display Mode: -15.60
```

When **TO_NUMBER** is used via a database driver, it also returns the type as NUMERIC with a scale of 2. In Logical mode or ODBC mode, the returned value is a canonical number, no scale is imposed on fractional digits, and trailing zeros are omitted.

SQL

```
SELECT TO_NUMBER('-15 degrees F') -- Logical/ODBC Mode: -15
```

SQL

```
SELECT TO_NUMBER('-15.835 degrees F') -- Logical/ODBC Mode: -15.835
```

SQL

```
SELECT TO_NUMBER('-15.60000 degrees F') -- Logical/ODBC Mode: -15.6
```

If the input string expression is of data type DOUBLE, then **TO_NUMBER** also returns the value as data type DOUBLE. All format modes display the full precision of the converted number.

SQL

```
SELECT TO_NUMBER(CAST('-15.6 degrees F' AS DOUBLE)) -- -15.599999999999999644
```

More About

Related SQL Functions

- **TO_NUMBER** converts a string to a number of data type NUMERIC.

- TO_CHAR performs the reverse operation; it converts a number to a string.

- CAST and CONVERT can be used to convert a string to a number of any data type. For example, you can convert a string to a number of data type INTEGER.

- TO_DATE converts a formatted date string to a date integer.

- TO_TIMESTAMP converts a formatted date and time string to a standard timestamp.

See Also

- Data Types

TO_POSIXTIME (SQL)

A date/time function that converts a formatted date string to a %PosixTime timestamp.

Synopsis

```
TO_POSIXTIME(date_string[,format])
```

```
TO_POSIXTIME(date_string[,format])
```

Description

The **TO_POSIXTIME** function converts date and time strings in various formats to a %PosixTime timestamp, with data type %Library.PosixTime. **TO_POSIXTIME** returns a %PosixTime timestamp as a calculated value based on the number of elapsed seconds from the arbitrary starting point of 1970-01-01 00:00:00, encoded as a 64-bit signed integer. The actual number of elapsed seconds (and fractional seconds) from this date is the Unix®timestamp, a numeric value. InterSystems IRIS encodes the Unix® timestamp to generate the %PosixTime timestamp. Because a %PosixTime timestamp value is encoded, 1970-01-01 00:00:00 is represented as 1152921504606846976. Dates prior to 1970-01-01 00:00:00 have a negative integer value. Refer to the %PosixTime data type for further details.

TO_POSIXTIME does not convert timezones; a local date and time is converted to a local %PosixTime timestamp; a UTC date and time is converted to a UTC %PosixTime timestamp.

The earliest date supported by %PosixTime is 0001-01-01 00:00:00, which has a logical value of -6979664624441081856. The last date supported is 9999-12-31 23:59:59.999999, which has a logical value of 1406323805406846975. These limits correspond to the ODBC date format display limits. These values can be further limited using the %Library.PosixTime MINVAL and MAXVAL parameters. You can use the **IsValid()** method to determine if a numeric value is a valid %PosixTime value.

A %PosixTime value always encodes a precision of 6 decimal digits of fractional seconds. A *date_string* with fewer digits of precision is zero-padded to 6 digits before %PosixTime conversion; a *date_string* with more than 6 digits of precision is truncated to 6 digits before %PosixTime conversion.

If *date_string* omits components of the timestamp, **TO_POSIXTIME** supplies the missing components. If both *date_string* and *format* omit the year, yyyy defaults to the current year; if only *date_string* omits the year, it defaults to 00, which is expanded to a four-digit year according to the year *format* element. If a day or month value is omitted, dd defaults to 01; mm-dd defaults to 01-01. A missing time component defaults to 00. Fractional seconds are supported, but must be explicitly specified; no fractional seconds are provided by default.

TO_POSIXTIME supports conversion of two-digit years to four digits. **TO_POSIXTIME** supports conversion of 12-hour clock time to 24-hour clock time. It provides range validation of date and time element values, including leap year validation. Range validation violations generate an SQLCODE -400 error.

This function can also be invoked from ObjectScript using the **TOPOSIXTIME()** method call:

```
$SYSTEM.SQL.Functions.TOPOSIXTIME(date_string,format)
```

The **TO_POSIXTIME** function can be used in data definition when supplying a default value to a field. For example:

```
CREATE TABLE mytest
(ID NUMBER(12,0) NOT NULL,
End_Year DATE DEFAULT TO_POSIXTIME('12-31-2018','MM-DD-YYYY') NOT NULL)
```

TO_POSIXTIME can be used with the **CREATE TABLE** or **ALTER TABLE ADD COLUMN** statements. Only a literal value for *date_string* can be used in this context. For further details, refer to the CREATE TABLE command.

%PosixTime Representation

%PosixTime encodes 6 digits of precision for fractional seconds, regardless of the precision of the *date_string*. The ODBC and Display modes truncate trailing zeros of precision.

• Logical Mode: an encoded 64-bit (19 characters) signed integer.

• ODBC Mode: YYYY–MM–DD HH:MM:SS.FFFFFF. Refer to the %PosixTime **LogicalToOdbc()** method.

• Display Mode: uses the default date/time formats (*dformat* -1 and *tformat* -1) for the current locale, as described in $ZDATETIME. Refer to the %PosixTime **LogicalToDisplay()** method.

Related SQL Functions

• **TO_POSIXTIME** converts a formatted date and time string to a %PosixTime timestamp.

• TO_CHAR performs the reverse operation; it converts a %PosixTime timestamp to a formatted date and time string.

• UNIX_TIMESTAMP converts a formatted date and time string to a Unix® timestamp.

• TO_DATE converts a formatted date string to a date integer.

• CAST and CONVERT perform %PosixTime data type conversion.

Arguments

date-string

A string expression to be converted to a %PosixTime timestamp. This expression may contain a date value, a time value, or a date and time value.

format

An optional date and time format string corresponding to *date_string*. If omitted, defaults to `DD MON YYYY HH:MI:SS`

Date and Time String

The *date_string* argument specifies a date and time string literal. If you supply a date string with no time component, **TO_POSIXTIME** supplies the time value 00:00:00. If you supply a time string with no date component, **TO_POSIXTIME** supplies the date of 01–01 (January 1) of the current year.

You can supply a date and time string of any kind for the input *date_string*. Each *date_string* character must correspond to the *format* string, with the following exceptions:

• Leading zeros may be included or omitted (with the exception a *date_string* without separator characters).

• Years may be specified with two digits or four digits.

• Month abbreviations (with *format* MON) must match the month abbreviation for that locale. For some locales, a month abbreviation may not be the initial sequential characters of the month name. Month abbreviations are not case-sensitive.

• Month names (with *format* MONTH) should be specified as full month names. However, **TO_POSIXTIME** does not require full month names with *format* MONTH; it accepts the initial character(s) of the full month name and selects the first month in the month list that corresponds to that initial letter sequence. Therefore, in English, "J" = "January", "Ju" = "June", "Jul" = "July". All characters specified must match the sequential characters of the full month name; characters beyond the full month name are not checked. For example, "Fe", "Febru", and "FebruaryLeap" are all valid values; "Febs" is not a valid value. Month names are not case-sensitive.

• Time values can be input with the time separator characters defined for the locale. The output timestamp always represents the time value with the ODBC standard time separator characters: colon (:) and period (.). An omitted time element defaults to zeroes.

Format

A *format* is a string of one or more format elements specified according to the following rules:

- Format elements are not case-sensitive.

- Almost any sequence or number of format elements is permitted.

- Format strings separate their elements with non-alphanumeric separator characters (for example, a space, slash, or hyphen) that match the separator characters in the *date_string*. These separator characters do not appear in the output string, which uses standard timestamp separators: hyphens for date values, colons for time values, and a period (when required) for fractional seconds. This use of separator characters does not depend on the DateSeparator defined for your NLS locale.

- The following date format strings do not require separator characters: MMDDYYYY, DDMMYYYY, YYYYMMDDHHMISS, YYYYMMDDHHMI, YYYYMMDDHH, YYYYMMDD, YYYYDDMM, HHMISS, and HHMI. The incomplete date format YYYYMM is also supported, and assume a DD value of 01. Note that in these cases leading zeros must be provided for all elements (such as MM and DD), with the exception of the final element.

- Characters in *format* that are not valid format elements are ignored.

Format Elements

The following table lists the valid date format elements for the *format* argument:

Element	Meaning
DD	Two-digit day of month (01-31). Leading zeros are not required, unless *format* contains no date separator characters.
MM	Two-digit month number (01-12; 01 = January). Leading zeros are not required, unless *format* contains no date separator characters. In Japanese and Chinese, a month number consists of a numeric value followed by the ideogram for "month".
MON	Abbreviated name of month, as specified by the MonthAbbr property in the current locale. By default, in English this is the first three letters of the month name. In other locales, month abbreviations may be more than three letters long and/or may not consist of the first letters of the month name. A period character is not permitted. Not case-sensitive.
MONTH	Full name of the month, as specified by the MonthName property in the current locale. Not case-sensitive.
YYYY	Four-digit year.
YY	Last two digits of the year. The first 2 digits of a YY 2-digit year default to 19.
RR / RRRR	Two-digit year to four-digit year conversion. (See below.)
DDD	Day of the year. The number of days since January 1. (See below.)
HH	Hour, specified as either 01–12 or 00–23, depending on whether a meridian indicator (AM or PM) is specified. Can be specified as HH12 or HH24.
MI	Minute, specified as 00–59.
SS	Second, specified as 00–59.

Element	Meaning
FF	Fractions of a second. FF indicates that one or more fractional digits are provided; *date_string* can specify any number of fractional digits. **TO_POSIXTIME** returns exactly six digits of precision, regardless of the precision supplied in *date_string*.
AM / PM	Meridian indicator, specifies a 12–hour clock. (See below.)
A.M. / P.M.	Meridian indicator (with periods), specifies a 12–hour clock. (See below.)

A **TO_POSIXTIME** *format* can also include a D (day of week number), DY (day of week abbreviation), or DAY (day of week name) element to match the input *date_string*. However, these format elements are not validated or used to determine the return value. For further details on these *format* elements, refer to TO_CHAR.

Two-Digit Year Conversion (RR and RRRR formats)

The RR format provides two-digit to four-digit year conversion. **TO_POSIXTIME** performs this conversion using the default date format (*dformat* -1), which uses the YearOption property of current locale, as described in $ZDATETIME.

Day of the Year (DDD format)

You can use DDD to convert the day of the year (number of days elapsed since January 1) to an actual date. The format string DDD YYYY must be paired with a corresponding *date_string* consisting of an integer number of days and a four-digit year. (Two-digit years must be specified as RR (not YY) when used with DDD.) The format string DDD defaults to the current year. The number of elapsed days must be a positive integer in the range 1 through 365 (366 if YYYY is a leap year). The four-digit year must be within the standard InterSystems IRIS date range: 1841 through 9999. (If you omit the year, it defaults to the current year.) The DDD and year (YYYY, RRRR, or RR) format elements can be specified in any order; a separator character between them is mandatory; this separator can be a blank space. The following example shows this use of Day of the Year:

SQL

```
SELECT TO_POSIXTIME('2018:160','YYYY:DDD')
```

If a format string contains both a DD and a DDD element, the DDD element is dominant. This is shown in the following example, which returns 2008-02-29 00:00:00 (not 2008-12-31 00:00:00):

SQL

```
SELECT TO_POSIXTIME('2018-12-31-60','YYYY-MM-DD-DDD')
```

TO_POSIXTIME permits you to return a date expression corresponding to a day of the year. **TO_CHAR** permits you to return the day of the year corresponding to a date expression.

Dates Before 1970

TO_POSIXTIME represents a date before January 1, 1970 as a negative number. %PosixTime cannot represent dates before January 1, 0001 or after December 31, 9999. Attempted to input such a date results in an SQLCODE -400 error. The **TO_DATE** function provides a Julian date format to represent BCE dates before January 1, 0001. Julian date conversion converts a seven-digit internal positive integer value (a Julian day count) to a display-format or ODBC-format date. Time values are not supported for Julian dates.

12-Hour Clock Time

A %PosixTime timestamp always represents time using a 24-hour clock. A *date_string* may represent time using a 12-hour clock or a 24-hour clock. **TO_POSIXTIME** assumes a 24-hour clock, unless one of the following applies:

- The *date_string* time value is followed by 'am' or 'pm' (with no periods). These meridian indicators are not case-sensitive, and may be appended to the time value, or be separated from it by one or more spaces.

- The *format* follows the time format with an 'a.m.' or 'p.m.' element (either one), separated from the time format by one or more spaces. For example: `DD MON YYYY HH:MI:SS.FF P.M.` This *format* supports 12-hour clock *date_string* values such as `2:23pm`, `2:23:54.6pm`, `2:23:54 pm`, `2:23:54 p.m.`, and `2:23:54` (assumed to be AM). Meridian indicators are not case-sensitive. When using a meridian indicator with periods, it must be separated from the time value by one or more spaces.

Examples

The following Embedded SQL example converts the current local datetime to a %PosixTime value. (Note that *format* uses "ff" to represent any number of fractional digits; in this case, 3 digits of precision. %PosixTime encodes this as 6 digits of precision, supplying three trailing zeroes.) This example then uses the %Posix **LogicalToOdbc()** method to display this value as an ODBC timestamp, trimming trailing zeroes of precision:

ObjectScript

```
SET tstime=$ZDATETIME($ZTIMESTAMP,3,1,3)
WRITE "local datetime in : ",tstime,!
&sql(SELECT
     TO_POSIXTIME(:tstime,'yyyy-mm-dd hh:mi:ss.ff')
     INTO :ptime)
IF SQLCODE=0 {
  WRITE "Posix encoded datetime: ",ptime,!
  SET ODBCout=##class(%PosixTime).LogicalToOdbc(ptime)
  WRITE "local datetime out: ",ODBCout   }
ELSE { WRITE "SQLCODE error:",SQLCODE }
```

The following example specifies date strings in various formats. The first one uses the default format, the others specify a *format*. All of these convert *date_string* to the timestamp value of `2018-06-29 00:00:00`:

SQL

```
SELECT
     TO_POSIXTIME('29 JUN 2018'),
     TO_POSIXTIME('2018 Jun 29','YYYY MON DD'),
     TO_POSIXTIME('JUNE 29, 2018','month dd, YYYY'),
     TO_POSIXTIME('2018***06***29','YYYY***MM***DD'),
     TO_POSIXTIME('06/29/2018','MM/DD/YYYY'),
     TO_POSIXTIME('29/6/2018','DD/MM/YYYY')
```

The following example specifies the YYYYMM date format. It does not require element separators. **TO_POSIXTIME** supplies the missing day and time values:

SQL

```
 SELECT TO_POSIXTIME('201806','YYYYMM')
```

This example returns the timestamp `2018-06-01 00:00:00`.

The following example specifies just the HH:MI:SS.FF time format. **TO_POSIXTIME** supplies the missing date value. In each case, this example returns the date of `2018-01-01` (where 2018 is the current year):

SQL

```
SELECT TO_POSIXTIME('11:34','HH:MI:SS.FF'),
     TO_POSIXTIME('11:34:22','HH:MI:SS.FF'),
     TO_POSIXTIME('11:34:22.00','HH:MI:SS.FF'),
     TO_POSIXTIME('11:34:22.7','HH:MI:SS.FF'),
     TO_POSIXTIME('11:34:22.7000000','HH:MI:SS.FF')
```

Note that fractional seconds are passed through exactly as specified, with no padding or truncation.

See Also

- SQL commands: CREATE TABLE, ALTER TABLE

- SQL functions: CAST, CONVERT, TO_CHAR, TO_DATE, TO_NUMBER, TO_TIMESTAMP, UNIX_TIMESTAMP

- ObjectScript functions: $ZDATETIME, $ZDATETIMEH

- ObjectScript special variable: $ZTIMESTAMP

TO_TIMESTAMP (SQL)

A date function that converts a formatted string to a timestamp.

Synopsis

```
TO_TIMESTAMP(dateString,format)
TO_TIMESTAMP(dateString)
```

Description

The **TO_TIMESTAMP** function converts date and time strings in various formats to the standard InterSystems IRIS® representation of a timestamp. The returned timestamp is of data type TIMESTAMP and has this format, with leading zeros included and a 24-hour clock time:

```
yyyy-mm-dd hh:mi:ss
```

The returned timestamp includes leading zeros and uses a 24-hour clock time by default.

TO_TIMESTAMP supports fractional seconds, two-digit to four-digit year conversions, and 12-hour to 24-hour clock time conversions. It provides range validation of date and time element values, including leap year validation. Range validation violations generate an SQLCODE -400 error.

TO_TIMESTAMP returns a standard timestamp in ODBC format. To return an encoded 64-bit timestamp, use TO_POSIXTIME instead. For details on other SQL functions that perform conversions, see Related SQL Functions.

- **TO_TIMESTAMP(*dateString*,*format*)** converts the date string using the specified format string. The date and time elements of *dateString* must be compatible with the format elements in the corresponding positions of *format*.

  This statement converts a string that specifies the date from the previous syntax in a different format.

  SQL

  ```
  SELECT TO_TIMESTAMP('June 29, 2022 12:34 PM', 'MONTH DD, YYYY')
  ```

  Examples:

  – Convert Date Strings to Multiple Timestamp Formats

  – Set Default Timestamp Column Values

  – Two-Digit Year Conversion (RR and RRRR Formats)

  – Day of the Year Conversion (DDD Format)

- **TO_TIMESTAMP(*dateString*)** converts the date string using the default format of DD MON YYYY HH:MI:SS. The *dateString* argument must be compatible with this format. If you omit the time, or specify only a portion of the time, **TO_TIMESTAMP** returns the time as 00:00:00.

  This statement converts the date string 29 Jun 2022 to the timestamp 2022-06-29 12:34:00.

  SQL

  ```
  SELECT TO_TIMESTAMP('29 Jun 2022 12:34')
  ```

  Example: Convert Date Strings to Multiple Timestamp Formats

Arguments

dateString

The *dateString* argument is a string expression that specifies the date string to be converted to a timestamp. *dateString* can contain a date value, a time value, or both.

Each character of *dateString* must correspond to the *format* string, with the following exceptions:

- Leading zeros can be included or omitted, with the exception of a *dateString* without separator characters.

- Years can be specified with two digits or four digits.

- Month abbreviations (with *format* MON) must match the month abbreviation for that locale. For some locales, a month abbreviation may not be the initial sequential characters of the month name. Month abbreviations are not case-sensitive.

- Month names (with *format* MONTH) should be specified as full month names. However, **TO_TIMESTAMP** does not require full month names to match *format* MONTH. It accepts the initial characters of the full month name and selects the first month in the month list that corresponds to that initial letter sequence. Therefore, in English, "J" = "January", "Ju" = "June", and "Jul" = "July". All characters specified must match the sequential characters of the full month name. Characters beyond the full month name are not checked. For example, "Fe", "Febru", and "FebruaryLeap" are all valid values, but "Febs" is not. Month names are not case-sensitive.

- Time values can be specified with the time separator characters defined for the locale. The output timestamp always represents the time value with the ODBC standard time separator characters: colon (:) for hours, minutes, and seconds, and period (.) for fractional seconds. An omitted time element defaults to zeros. By default, a timestamp is returned without fractional seconds.

If *dateString* omits components of the timestamp, **TO_TIMESTAMP** supplies the missing components:

- If you specify a date without a time, **TO_TIMESTAMP** sets the returned time value to 00:00:00. If you omit only a portion of the time, **TO_TIMESTAMP** sets that portion to 00. Fractional seconds are supported but must be explicitly specified.

- If you specify a time without a date, **TO_TIMESTAMP** sets the returned date value to 01-01 (January 1) of the current year. If you omit only the day, the day defaults to 01. If you also omit the month, the month and day default to 01-01

- If you omit the year, the year defaults to 00, which is expanded to a four-digit year according to the year element of *format*. If you also omit the year in *format*, YYYY defaults to the current year.

You can specify **TO_TIMESTAMP** dates from January 1, 0001 to December 31, 9999. To represent earlier dates, use the TO_DATE function.

format

The *format* argument is a date and time string that specifies the format of the *dateString*. **TO_TIMESTAMP** converts the date elements of *dateString* according to the format elements in the corresponding positions of *format*. The elements of *format* follow these rules:

- Format elements are not case-sensitive.

- Almost any sequence or number of format elements is permitted.

- Format strings separate their elements with non-alphanumeric separator characters (for example, a space, slash, or hyphen) that match the separator characters in the *dateString*. These separator characters do not appear in the output string, which uses standard timestamp separators: hyphens for date values, colons for time values, and a period (when required) for fractional seconds. This use of separator characters does not depend on the DateSeparator defined for your NLS (National Language Support) locale.

- The following date format strings do not require separator characters: `MMDDYYYY`, `DDMMYYYY`, `YYYYMMDDHHMISS`, `YYYYMMDDHHMI`, `YYYYMMDDHH`, `YYYYMMDD`, `YYYYDDMM`, `HHMISS`, and `HHMI`. The incomplete date format `YYYYMM` is also supported and assumes a DD value of 01. In these formats, you specify leading zeros for all elements, with the exception of the final element.

- Invalid format elements in *format* are ignored.

This table lists the valid date and time format elements. The uppercase form is shown, but these elements are not case-sensitive.

Element	Meaning
DD	Two-digit day of month (01-31). Leading zeros are not required, unless *format* contains no date separator characters.
MM	Two-digit month number (01-12; 01 = January). Leading zeros are not required, unless *format* contains no date separator characters. In Japanese and Chinese, a month number consists of a numeric value followed by the ideogram for "month".
MON	Abbreviated name of month, as specified by the MonthAbbr property in the current locale. By default, in English, this is the first three letters of the month name. In other locales, month abbreviations can be more than three letters long and might not consist of the first letters of the month name. A period character is not permitted. Not case-sensitive.
MONTH	Full name of the month, as specified by the MonthName property in the current locale. Not case-sensitive.
YYYY	Four-digit year in the range 0001 to 9999.
YY	Last two digits of the year. The first two digits of a two-digit year default to 19.
RR / RRRR	Two-digit year to four-digit year conversion. For more details, see Two-Digit Year Conversion (RR and RRRR Formats).
DDD	Day of the year. The count of days since January 1. For more details, Day of the Year Conversion (DDD Format).
HH	Hour (1–12 or 00–23), depending on whether a meridian indicator (AM or PM) is specified. Can be specified as HH12 or HH24.
MI	Minute, specified as 00–59.
SS	Second, specified as 00–59.
FF	Fractions of a second. **TO_TIMESTAMP** returns the exact fractional value specified in *dateString*, without padding or truncating the value. To specify FF, you must provide a decimal separator format character (.). If you do not specify FF, fractional seconds specified in *dateString* are ignored.

Element	Meaning
AM	Meridian indicator specifying a 12-hour clock time.
PM A.M. P.M.	The TIMESTAMP data type always represents time using a 24-hour clock. A *dateString* can represent time using a 12-hour clock or a 24-hour clock. **TO_TIMESTAMP** assumes a 24-hour clock, unless the time part of *dateString* ends with a meridian indicator. For example: `DD MON YYYY HH:MI:SS.FF P.M.` This format supports 12-hour clock values specified in *dateString* such as `2:23pm`, `2:23:54.6pm`, `2:23:54 pm`, `2:23:54 p.m.`, and `2:23:54` (assumed to be AM). If you specify periods in the meridian indictors, you must separate the indicator from the time with at least one space.

A **TO_TIMESTAMP** *format* can also include a D (day of week number), DY (day of week abbreviation), or DAY (day of week name) element. However, these format elements are not validated or used to determine the return value. For more details on these *format* elements, see TO_CHAR.

If you omit *format*, **TO_TIMESTAMP** parses the date string using the default format of `DD MON YYYY HH:MI:SS`. For example, `'01 Feb 3456 07:08:09'`.

Examples

Convert Date Strings to Multiple Timestamp Formats

This statement specifies date strings in various formats. The first one uses the default format, the others specify a format argument that **TO_TIMESTAMP** uses to parse the date string. **TO_TIMESTAMP** converts all these date strings to the timestamp `2022-06-29 00:00:00`.

SQL

```
SELECT
  TO_TIMESTAMP('29 JUN 2022'),
  TO_TIMESTAMP('2022 Jun 29','YYYY MON DD'),
  TO_TIMESTAMP('JUNE 29, 2022','month dd, YYYY'),
  TO_TIMESTAMP('2022***06***29','YYYY***MM***DD'),
  TO_TIMESTAMP('06/29/2022','MM/DD/YYYY'),
  TO_TIMESTAMP('29/6/2022','DD/MM/YYYY')
```

This statement specifies the YYYYMM date format. It does not require element separators. **TO_TIMESTAMP** supplies the missing day and time values and returns the timestamp `2022-06-01 00:00:00`.

SQL

```
SELECT TO_TIMESTAMP('202206','YYYYMM')
```

This statement specifies just the HH:MI:SS.FF time format. **TO_TIMESTAMP** supplies the missing date value, returning in all cases a date value of `YYYY-01-01`, where *YYYY* is the current year. The time value varies based on the fractional seconds specified in the date string. **TO_TIMESTAMP** passes fractional seconds through exactly as specified, with no padding or truncation.

SQL

```
SELECT TO_TIMESTAMP('11:34','HH:MI:SS.FF'),
  TO_TIMESTAMP('11:34:22','HH:MI:SS.FF'),
  TO_TIMESTAMP('11:34:22.00','HH:MI:SS.FF'),
  TO_TIMESTAMP('11:34:22.7','HH:MI:SS.FF'),
  TO_TIMESTAMP('11:34:22.7000000','HH:MI:SS.FF')
```

This statement shows other ways to specify a time format with fractional seconds. All three calls to **TO_TIMESTAMP** return an ODBC-format timestamp with the time portion value as `11:34:22.9678`. In the first two calls, the omitted date portion defaults to January 1 of the current year. The third call specifies a date portion.

SQL

```
SELECT TO_TIMESTAMP('113422.9678','HHMISS.FF'),
       TO_TIMESTAMP('9678.113422','FF.HHMISS'),
       TO_TIMESTAMP('9678.20220629113422','FF.YYYYMMDDHHMISS')
```

Set Default Timestamp Column Values

TO_TIMESTAMP can supply a default timestamp value to columns in a table. For example, this statement creates a table that accepts default values for `ReviewDate`, a column of type TIMESTAMP.

```
CREATE TABLE Sample.MyEmpReviews (
  EmpNum INTEGER UNIQUE NOT NULL,
  ReviewDate TIMESTAMP DEFAULT TO_TIMESTAMP(365,'DDD'))
```

If you insert a row without specifying a `ReviewDate` value, then the `ReviewDate` is set to the default timestamp of the 365th day of the current year.

SQL

```
INSERT INTO Sample.MyEmpReviews (EmpNum) VALUES (1)
```

You can use **TO_TIMESTAMP** to set default column values in both **CREATE TABLE** and **ALTER TABLE** **ADD COLUMN** statements. When setting these defaults, *dateString* must be a literal value.

Two-Digit Year Conversion (RR and RRRR Formats)

The YY format converts a two-digit year value to four digits by appending 19. For example, 07 becomes 1907 and 93 becomes 1993. The RR and RRRR formats provide more flexible two-digit to four-digit year conversions.

The RR format conversion is based on the current year.

- If the current year is in the first half of a century:

  - Two-digit years from 00 through 49 are expanded to a four-digit year in the current century.

  - Two-digit years from 50 through 99 are expanded to a four-digit year in the previous century.

  This statement shows the display format of timestamps that **TO_TIMESTAMP** returns when the current year is between 2000 and 2050.

  SQL

  ```
  SELECT
    TO_TIMESTAMP('29 September 00','DD MONTH RR'), -- 2000-09-29 00:00:00
    TO_TIMESTAMP('29 September 18','DD MONTH RR'), -- 2018-09-29 00:00:00
    TO_TIMESTAMP('29 September 49','DD MONTH RR'), -- 2049-09-29 00:00:00
    TO_TIMESTAMP('29 September 50','DD MONTH RR'), -- 1950-09-29 00:00:00
    TO_TIMESTAMP('29 September 77','DD MONTH RR')  -- 1977-09-29 00:00:00
  ```

- If the current year is in the second half of a century, all two-digit years are expanded to a four-digit year in the current century.

  This statement shows the display format of dates that **TO_TIMESTAMP** returns when the current year is between 2050 and 2099.

SQL

```
SELECT
    TO_TIMESTAMP('29 September 00','DD MONTH RR'),  -- 2000-09-29 00:00:00
    TO_TIMESTAMP('29 September 21','DD MONTH RR'),  -- 2021-09-29 00:00:00
    TO_TIMESTAMP('29 September 49','DD MONTH RR'),  -- 2049-09-29 00:00:00
    TO_TIMESTAMP('29 September 50','DD MONTH RR'),  -- 1950-09-29 00:00:00
    TO_TIMESTAMP('29 September 77','DD MONTH RR')   -- 1977-09-29 00:00:00
```

Using the RRRR format, you can input a mix of two-digit and four-digit years. **TO_TIMESTAMP** passes four-digit years through unchanged. **TO_TIMESTAMP** converts two-digit years to four-digit years by using the RR format algorithm described earlier in this example.

This statement shows the display format of dates that **TO_TIMESTAMP** returns when the current year is between 2000 and 2050.

SQL

```
SELECT
  TO_TIMESTAMP('29 September 2021','DD MONTH RRRR'),  -- 2021-09-29 00:00:00
  TO_TIMESTAMP('29 September 21','DD MONTH RRRR'),    -- 2021-09-29 00:00:00
  TO_TIMESTAMP('29 September 1949','DD MONTH RRRR'),  -- 1949-09-29 00:00:00
  TO_TIMESTAMP('29 September 49','DD MONTH RRRR'),    -- 2049-09-29 00:00:00
  TO_TIMESTAMP('29 September 1950','DD MONTH RRRR'),  -- 1950-09-29 00:00:00
  TO_TIMESTAMP('29 September 50','DD MONTH RRRR')     -- 1950-09-29 00:00:00
```

Day of the Year Conversion (DDD Format)

You can use the DDD format to convert the day of the year (that is, the number of days elapsed since January 1) to an actual timestamp. To perform this conversion:

*   The date portion of the *format* argument must contain the DDD format element and optionally a year format such as YYYY, YY, RR, or RRRR. You can specify these elements in any order but they must include a separator character between them. If you omit the year element, then **TO_TIMESTAMP** defaults to the current year.

*   The *dateString* argument must contain corresponding *day* and *year* values, where:

    *   *day* is an integer in the range 1 through 365 (366 if *year* is a leap year).

    *   *year* is a year in the range 0001 through 9999.

This statement returns the 60th day of the year 2022.

SQL

```
SELECT TO_TIMESTAMP('2022:60','YYYY:DDD') --2022-03-01 00:00:00
```

TO_TIMESTAMP passes month elements through unchanged. If a format string contains both a DD and a DDD element, **TO_TIMESTAMP** processes the DDD element and ignores the DD element. For example, this statement returns a timestamp for the date 2/29/2020 (the 60th day of 2020), not for 12/31/2020:

SQL

```
SELECT TO_TIMESTAMP('2020-12-31-60','YYYY-MM-DD-DDD')
```

TO_TIMESTAMP returns a timestamp expression containing the day of the year, not the day of the year itself. To return this day value, use **TO_CHAR**.

More About

Related SQL Functions

- **TO_TIMESTAMP** converts a formatted date and time string to a standard timestamp.

- TO_CHAR performs the reverse operation. It converts a standard timestamp to a formatted date and time string.

- TO_DATE converts a formatted date string to a date integer.

- CAST and CONVERT perform TIMESTAMP data type conversion.

Alternatives

To perform equivalent timestamp conversions in ObjectScript, use the **TOTIMESTAMP()** method:

```
$SYSTEM.SQL.Functions.TOTIMESTAMP(date_string,format)
```

See Also

- SQL commands: CREATE TABLE, ALTER TABLE

- SQL functions: CAST, CONVERT, TO_CHAR, TO_DATE, TO_NUMBER, TO_POSIXTIME

- ObjectScript functions: $ZDATE $ZDATEH

$TRANSLATE (SQL)

A string function that performs character-for-character replacement.

Synopsis

```
$TRANSLATE(string,identifier[,associator])
```

Description

The **$TRANSLATE** function performs character-for-character replacement within a return value string. It processes the *string* argument one character at a time. It compares each character in *string* with each character in the *identifier* argument. If **$TRANSLATE** finds a match, it makes note of the position of that character.

- The two-argument form of **$TRANSLATE** removes all instances of the characters in the *identifier* argument from the output string.

- The three-argument form of **$TRANSLATE** replaces all instances of each *identifier* character found in the *string* with the positionally corresponding *associator* character. Replacement is performed on a character, not a string, basis. If the *identifier* argument contains more characters than the *associator* argument, the excess characters in the *identifier* argument are deleted from the output string. If the *identifier* argument contains fewer characters than the *associator* argument, the excess character(s) in the *associator* argument are ignored.

$TRANSLATE is case-sensitive.

$TRANSLATE cannot be used to replace NULL with a character.

SQLCODE -380 is issued if you specify too few arguments. SQLCODE -381 is issued if you specify too many arguments.

$TRANSLATE and REPLACE

$TRANSLATE performs character-for-character matching and replacement. REPLACE performs string-for-string matching and replacement. **REPLACE** can replace a single specified substring of one or more characters with another substring, or remove multiple instances of a specified substring. **$TRANSLATE** can replace multiple specified characters with corresponding specified replacement characters.

By default, both functions are case-sensitive, start at the beginning of *string*, and replace all matching instances. **REPLACE** has arguments that can be used to change these defaults.

Arguments

string

The target string. It can be a field name, a literal, a host variable, or an SQL expression.

identifier

The character(s) to search for in *string*. It can be a string or numeric literal, a host variable, or an SQL expression.

associator

An optional argument. The replacement character(s) corresponding to each character in the *identifier*. It can be a string or numeric literal, a host variable, or an SQL expression.

Examples

In the following example, a two-argument **$TRANSLATE** modifies Name values by removing punctuation (commas, spaces, periods, apostrophes, hyphens), returning names that consist of only alphabetic characters. Note that the *identifier* doubles the apostrophe to escape it as a literal character, rather than a string delimiter:

SQL

```
SELECT TOP 20 Name,$TRANSLATE(Name,', .''-') AS AlphaName
FROM Sample.Person
WHERE Name %STARTSWITH 'O'
```

In the following example, a three-argument **$TRANSLATE** modifies Name values by replacing commas and spaces with caret (^) characters, returning names delimited in three pieces (surname, first name, middle initial). Note that the *associator* must specify "^" as many times as the number of characters in *identifier*:

SQL

```
SELECT TOP 20 Name,$TRANSLATE(Name,', ','^^') AS PiecesNamePunc
FROM Sample.Person
WHERE Name %STARTSWITH 'O'
```

In the following example, a three-argument **$TRANSLATE** modifies Name values by both replacing commas and spaces with caret (^) characters (specified in the *identifier* and *associator*) and removing periods, apostrophes, and hyphens (specified in the *identifier*, omitted from the *associator*):

SQL

```
SELECT TOP 20 Name,$TRANSLATE(Name,', .''-','^^') AS PiecesNameNoPunc
FROM Sample.Person
WHERE Name %STARTSWITH 'O'
```

See Also

- REPLACE function

- STUFF function

- String Manipulation

TRIM (SQL)

A string function that returns a character string with specified leading and/or trailing characters removed.

Synopsis

```
TRIM([end_keyword] [characters FROM] string-expression)
```

Description

TRIM strips the specified characters from the beginning and/or end of a supplied value. By default, stripping of letters is case-sensitive. Character stripping from either end stops when a character not specified in *characters* is encountered. The default is to strip blank spaces from both ends of *string-expression*.

TRIM always returns data type VARCHAR, regardless of the data type of the input expression to be trimmed.

Note that leading zeros are automatically stripped from numbers before they are supplied to **TRIM** or any other SQL function. To retain leading zeros, a number must be specified as a string.

The optional *end_keyword* argument can take the following values:

LEADING	A keyword that specifies that the characters in *characters* are to be removed from the beginning of *string-expression*.
TRAILING	A keyword that specifies that the characters in *characters* are to be removed from the end of *string-expression*.
BOTH	A keyword that specifies that the characters in *characters* are to be removed from both the beginning and end of *string-expression*. BOTH is the default and is used if no *end_keyword* is specified.

Alternatively, you can use **LTRIM** to trim leading blanks, or **RTRIM** to trim trailing blanks.

To pad a string with leading or trailing blanks or other characters, use **LPAD** or **RPAD**.

You can use the **LENGTH** function to determine if blank spaces have been stripped from or added to a string.

Characters to Strip

• All characters: **TRIM** returns an empty string if *characters* contains all the characters in *string-expression*.

• Single quote characters: **TRIM** can trim single-quote characters if these characters are doubled in both *characters* and *string-expression*. Thus, TRIM(BOTH 'a''b' FROM 'bb''ba''acaaa''') returns 'c'.

• Blank spaces: **TRIM** trims blank spaces from *string-expression* if *characters* is omitted. If *characters* is specified, it must include the blank space character to strip blank spaces.

• %List: If *string-expression* is a %List, **TRIM** can only trim trailing characters, not leading characters. This is because a %List contains leading encoding characters. You must convert a %List to a string to apply **TRIM** to leading characters.

• NULL: **TRIM** returns NULL if either string expression is NULL.

Arguments

end_keyword

An optional keyword specifying the which end of *string-expression* to strip. Available values are LEADING, TRAILING, or BOTH. The default is BOTH.

characters

An optional string expression specifying the characters to strip from *string-expression*. Every instance of the specified character(s) is stripped from the specified end(s) until a character not specified here is encountered. Thus `TRIM(BOTH 'ab' FROM 'bbbaacaaa')` returns 'c'. In this example, the BOTH keyword is optional.

If *characters* is not specified, **TRIM** strips blank spaces.

The FROM keyword is required if *characters* is specified. The FROM keyword is permitted (but not required) if *end_keyword* is specified and *characters* is not specified. If neither of these arguments are specified, the FROM keyword is not permitted.

string-expression

The string expression to be stripped. A *string-expression* can be the name of a column, a string literal, or the result of another function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR2).

The FROM keyword is omitted if both *characters* and *end_keyword* are omitted.

Examples

The following example uses the *end_keyword* and *characters* defaults; it removes leading and trailing blanks from "abc". The select-items concatenate '^' to both ends of the string to show blanks.

SQL

```
SELECT '^'||'  abc  '||'^' AS UnTrimmed,'^'||TRIM('  abc  ')||'^' AS Trimmed
```

returns the strings ^ abc ^ and ^abc^.

The following examples are all valid syntax to strip leading blank spaces from *string-expression*:

```
SELECT TRIM(LEADING '  abc  '),TRIM(LEADING FROM '  def  '),TRIM(LEADING ' ' FROM '  ghi  ')
```

The following example removes the character "x" from the beginning of the string "xxxabcxxx", resulting in "abcxxx":

SQL

```
SELECT TRIM(LEADING 'x' FROM 'xxxabcxxx') AS Trimmed
```

The following examples both remove the character "x" from the beginning and end of "xxxabcxxx", resulting in "abc". The first specifies BOTH, the second takes BOTH as the default:

SQL

```
SELECT TRIM(BOTH 'x' FROM 'xxxabcxxx') AS Trimmed
```

SQL

```
SELECT TRIM('x' FROM 'xxxabcxxx') AS Trimmed
```

The following example removes all instances of the characters "xyz" from the end of "abcxzzxyyyyz", resulting in "abc":

SQL

```
SELECT TRIM(TRAILING 'xyz' FROM 'abcxzzxyyyyz') AS Trimmed
```

The following example trims FullName by stripping all of the letters in FirstName, returning the last name preceded by a blank space. For example FirstName/Fullname 'Fred'/'Fred Rogers' returns ' Rogers'. In this example, FirstName 'Annie' would strip 'Ann', 'Anne', 'Ani', 'Ain', 'Annee', or 'Annie' from LastName, but would not completely strip 'Anna' because **TRIM** is case-sensitive; only 'A', not 'a', would be stripped.

SQL

```
SELECT TRIM(LEADING FirstName FROM FullName) FROM Sample.Person
```

The following example removes the leading letters "B" or "R" from the FavoriteColors values. Note that you must convert a list to a string in order to apply **TRIM** to leading characters:

SQL

```
SELECT TOP 15 Name,FavoriteColors,
       TRIM(LEADING 'BR' FROM $LISTTOSTRING(FavoriteColors)) AS Trimmed
       FROM Sample.Person WHERE FavoriteColors IS NOT NULL
```

See Also

- SQL functions: LTRIM, RTRIM, LPAD, RPAD

- ObjectScript function: $ZSTRIP

TRUNCATE (SQL)

A scalar numeric function that truncates a number at a specified number of digits.

Synopsis

```
{fn TRUNCATE(numeric-expr,scale)}
```

Description

TRUNCATE truncates *numeric-expr* by truncating at the *scale* number of digits from the decimal point. It does not round numbers or add padding zeroes. Leading and trailing zeroes are removed before the **TRUNCATE** operation.

- If *scale* is a positive number, truncation is performed at that number of digits to the right of the decimal point. If *scale* is equal to or larger than the number of decimal digits, no truncation or zero filling occurs.

- If *scale* is zero, the number is truncated to a whole integer. In other words, truncation is performed at zero digits to the right of the decimal point; all decimal digits and the decimal point itself are truncated.

- If *scale* is a negative number, truncation is performed at that number of digits to the left of the decimal point. If *scale* is equal to or larger than the number of integer digits in the number, zero is returned.

- If *numeric-expr* is zero (however expressed: 00.00, -0, etc.) **TRUNCATE** returns 0 (zero) with no decimal digits, regardless of the *scale* value.

- If *numeric-expr* or *scale* is NULL, **TRUNCATE** returns NULL.

TRUNCATE can only be used as an ODBC scalar function (with the curly brace syntax).

ROUND can be used to perform a similar truncation operation on numbers. **TRIM** can be used to perform a similar truncation operation on strings.

TRUNCATE, ROUND, and $JUSTIFY

TRUNCATE and **ROUND** are numeric functions that perform similar operations; they both can be used to decrease the number of significant decimal or integer digits of a number. **ROUND** allows you to specify either rounding (the default), or truncation; **TRUNCATE** does not perform rounding. **ROUND** returns the same data type as *numeric-expr*; **TRUNCATE** returns *numeric-expr* as data type NUMERIC, unless *numeric-expr* is data type DOUBLE, in which case it returns data type DOUBLE.

TRUNCATE truncates to a specified number of fractional digits. If the truncation results in trailing zeros, these trailing zeros are preserved. However, if *scale* is larger than the number of fractional decimal digits in the canonical form of *numeric-expr*, **TRUNCATE** does not zero-pad.

ROUND rounds (or truncates) to a specified number of fractional digits, but its return value is always normalized, removing trailing zeros. For example, `ROUND(10.004,2)` returns 10, not 10.00.

Use $JUSTIFY if rounding to a fixed number of fractional digits is important — for example, when representing monetary amounts. **$JUSTIFY** returns the specified number of trailing zeros following the rounding operation. When the number of digits to round is larger than the number of fractional digits, **$JUSTIFY** zero-pads. **$JUSTIFY** also right-aligns the numbers, so that the DecimalSeparator characters align in a column of numbers. **$JUSTIFY** does not truncate.

Arguments

numeric-expr

The number to be truncated. A number or numeric expression.

scale

An expression that evaluates to an integer that specifies the number of places to truncate, counting from the decimal point. Can be zero, a positive integer, or a negative integer. If *scale* is a fractional number, InterSystems IRIS rounds it to the nearest integer.

TRUNCATE returns the DOUBLE, INTEGER, or NUMERIC data type.

- If *numeric-expr* is of type DOUBLE, **TRUNCATE** returns DOUBLE.

- If *numeric-expr* is of type INTEGER and *scale* is less than or equal to 0, **TRUNCATE** returns INTEGER.

- If *numeric-expr* is of type NUMERIC, or of type INTEGER and *scale* is greater than 0, **TRUNCATE** returns NUMERIC.

Examples

The following two examples both truncate a number to two decimal digits, where *scale* is specified as an integer:

SQL

```
SELECT {fn TRUNCATE(654.321888,2)}
```

SQL

```
SELECT {fn TRUNCATE(654.321888,2)}
```

Both examples return 654.32 (truncation to two decimal places).

The following example specifies a *scale* larger than the number of decimal digits:

SQL

```
SELECT {fn TRUNCATE(654.321000,9)}
```

It returns 654.321 (InterSystems IRIS removed the trailing zeroes before the truncation operation; no truncation or zero padding occurred).

The following example specifies a *scale* of zero:

SQL

```
SELECT {fn TRUNCATE(654.321888,0)}
```

It returns 654 (all decimal digits and the decimal point are truncated).

The following example specifies a negative *scale*:

SQL

```
SELECT {fn TRUNCATE(654.321888,-2)}
```

It returns 600 (two integer digits have been truncated and replaced by zeroes; note that no rounding has been done).

The following example specifies a negative *scale* as large as the integer portion of the number:

SQL

```
SELECT {fn TRUNCATE(654.321888,-3)}
```

It returns 0.

See Also

- SQL functions: $JUSTIFY, ROUND, RTRIM, TRIM,

- ObjectScript function: $NORMALIZE

%TRUNCATE (SQL)

A collation function that truncates a string to the specified length and applies EXACT collation.

Synopsis

```
%TRUNCATE(expression[,length])
```

Description

%TRUNCATE truncates *expression* to the specified length, then returns it in the EXACT collation sequence.

EXACT collation orders pure numeric values (values for which x=+x) in numeric order first, followed by all other characters in string collation sequence. The EXACT string collation sequence is the same as the ANSI-standard ASCII collation sequence: digits are collated before uppercase alphabetic characters and uppercase alphabetic characters are collated before lowercase alphabetic characters. Punctuation characters occur at several places in the sequence.

%TRUNCATE passes through NULLs unchanged.

%TRUNCATE is an InterSystems SQL extension and is intended for SQL lookup queries.

This function can also be invoked from ObjectScript using the **TRUNCATE()** method call:

ObjectScript

```
WRITE $SYSTEM.SQL.Functions.TRUNCATE("This long string",9)
```

Arguments

expression

A string expression, which can be the name of a column, a string literal, or the result of another function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR2). *expression* can be a subquery.

length

An optional argument denoting the truncation length, specified as an integer. The initial *length* characters of *expression* are returned. If you omit *length*, **%TRUNCATE** collation is identical to **%EXACT** collation. You can enclose *length* with double parentheses to suppress literal substitution: ((*length*)).

Examples

The following example uses **%TRUNCATE** to return the first four characters of Name values:

SQL

```
SELECT TOP 5 Name,%TRUNCATE(Name,4) AS ShortName
FROM Sample.Person
```

The following example applies **%TRUNCATE** to a subquery:

SQL

```
SELECT TOP 5 Name, %TRUNCATE((SELECT Name FROM Sample.Company),10) AS Company
FROM Sample.Person
```

The following example uses **%TRUNCATE** in the **GROUP BY** clause to create an alphabet list that returns the number of names that begin with each letter:

SQL

```
SELECT Name AS FirstLetter,COUNT(Name) AS NameCount
FROM Sample.Person GROUP BY %TRUNCATE(Name,1) ORDER BY Name
```

The following two examples show how **%TRUNCATE** performs EXACT collation. The ORDER BY in the first example truncates Home_Street to two characters. Because the first two characters of a street address are almost always numbers, the Home_Street fields are ordered in the numeric sequence of their first two numbers.

SQL

```
SELECT Name,Home_Street
FROM Sample.Person
ORDER BY %TRUNCATE(Home_Street,2)
```

The ORDER BY in the second example truncates Home_Street to four characters. Because the fourth character of some street addresses is not a number (a blank space, for example), the Home_Street values that begin with four (or more) numbers are ordered first in numeric sequence, then the Home_Street values that contain a non-numeric character within the first four characters are ordered in string sequence:

SQL

```
SELECT Name,Home_Street
FROM Sample.Person
ORDER BY %TRUNCATE(Home_Street,4)
```

See Also

- CREATE TABLE

- %STARTSWITH predicate

- %EXACT collation function

- %SQLSTRING collation function

- %TRUNCATE collation function

- Collation

$TSQL_NEWID (SQL)

A function that returns a globally unique ID.

Synopsis

```
$TSQL_NEWID()
```

Description

$TSQL_NEWID returns a globally unique ID (GUID). A GUID is used to synchronize databases on occasionally connected systems. A GUID is a 36-character string consisting of 32 hexadecimal digits separated into five groups by hyphens. Its data type is %Library.UniqueIdentifier.

$TSQL_NEWID is provided in InterSystems SQL to support InterSystems Transact-SQL (TSQL). The corresponding TSQL function is NEWID.

The **$TSQL_NEWID** function takes no arguments. Note that the argument parentheses are required.

The %Library.GUID abstract class provides support for globally unique IDs, including the **AssignGUID()** method, which can be used to assign a globally unique ID to a class. To generate a GUID value, use the **%SYSTEM.Util.CreateGUID()** method.

Examples

The following example returns a GUID:

SQL

```
SELECT $TSQL_NEWID()
```

See Also

- TSQL: NEWID function

UCASE (SQL)

A case-transformation function that converts all lowercase letters in a string to uppercase letters.

Synopsis

```
UCASE(string-expression)
```

```
{fn UCASE(string-expression)}
```

Description

UCASE converts lowercase letters to uppercase for display purposes. It has no effects on non-alphabetic characters; it leaves numbers, punctuation, and leading or trailing blank spaces unchanged.

Note that **UCASE** can be used as an ODBC scalar function (with the curly brace syntax) or as an SQL general function.

UCASE does not force a numeric to be interpreted as a string. InterSystems SQL removes leading and trailing zeros from numerics. A numeric specified as a string retains leading and trailing zeros.

UCASE does not affect collation. The **%SQLUPPER** function is the preferred way in SQL to convert a data value for not case-sensitive collation. Refer to %SQLUPPER for further information on case transformation for collation.

This function can also be invoked from ObjectScript using the **UPPER()** method call:

```
$SYSTEM.SQL.UPPER(expression)
```

Arguments

string-expression

The string whose characters are to be converted to uppercase. The expression can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR).

Examples

The following example returns each person's name in uppercase letters:

SQL

```
SELECT Name,{fn UCASE(Name)} AS CapName
    FROM Sample.Person
```

UCASE also works on Unicode (non-ASCII) alphabetic characters, as shown in the following example, which converts Greek letters from lowercase to uppercase:

SQL

```
SELECT UCASE($CHAR(950,949,965,963))
```

See Also

- SQL functions: LCASE, %SQLUPPER, UPPER

- ObjectScript function: $ZCONVERT

UNIX_TIMESTAMP (SQL)

A date/time function that converts a date expression to a UNIX timestamp.

Synopsis

```
UNIX_TIMESTAMP([date-expression])
```

Description

UNIX_TIMESTAMP returns a UNIX® timestamp, the count of seconds (and fractional seconds) since '1970-01-01 00:00:00'.

If you do not specify *date-expression*, *date-expression* defaults to the current UTC timestamp. Therefore, `UNIX_TIMESTAMP()` is equivalent to `UNIX_TIMESTAMP(GETUTCDATE(3))`, assuming the system-wide default precision of 3.

If you specify *date-expression*, **UNIX_TIMESTAMP** converts the specified *date-expression* value to a UNIX timestamp, calculating the count of seconds to that timestamp. **UNIX_TIMESTAMP** can return a positive or negative count of seconds.

UNIX_TIMESTAMP returns its value as data type %Library.Numeric. It can return fractional seconds of precision. If you do not specify *date-expression*, it takes the currently configured system-wide precision. If you specify *date-expression* it takes its precision from *date-expression*.

date-expression Values

The optional *date-expression* can be specified as:

- An ODBC timestamp value (data type %Library.TimeStamp): YYYY-MM-DD HH:MI:SS.FFF

- A PosixTime timestamp value (data type %Library.PosixTime): an encoded 64-bit signed integer.

- A $HOROLOG date value (data type %Library.Date): a count of the number of days since December 31, 1840, where day 1 is January 1, 1841.

- A $HOROLOG timestamp, with or without fractional seconds: 64412,54736.

UNIX_TIMESTAMP does not perform timezone conversion: if *date-expression* is in UTC time, UTC UnixTime is returned; if *date-expression* is local time, a local UnixTime value is returned.

Fractional Seconds Precision

Fractional seconds are always truncated, not rounded, to the specified precision.

- A *date-expression* in %Library.TimeStamp data type format can have a maximum precision of nine. The actual number of digits supported is determined by the *date-expression precision* argument, the configured default time precision, and the system capabilities. If you specify a *precision* larger than the configured default time precision, the additional digits of precision are returned as trailing zeros.

- A *date-expression* in %Library.PosixTime data type format has a maximum precision of six. Every POSIXTIME value is computed using six digits of precision; these fractional digits default to zeros unless supplied.

Configuring Precision

The default precision can be configured using the following:

- SET OPTION with the TIME_PRECISION option.

- The system-wide **$SYSTEM.SQL.Util.SetOption()** method configuration option `DefaultTimePrecision`. To determine the current setting, call **$SYSTEM.SQL.CurrentSettings()** which displays `Default time precision`; the default is 0.

- Go to the Management Portal, select **System Administration**, **Configuration**, **SQL and Object Settings**, **SQL**. View and edit the current setting of **Default time precision for GETDATE(), CURRENT_TIME, and CURRENT_TIMESTAMP**.

Specify an integer 0 through 9 (inclusive) for the default number of decimal digits of precision to return. The default is 0. The actual precision returned is platform dependent; *precision* digits in excess of the precision available on your system are returned as zeroes.

Date and Time Functions Compared

UNIX_TIMESTAMP returns date and time expressed as a number of elapsed seconds from an arbitrary date.

TO_POSIXTIME returns an encoded 64-bit signed (a %PosixTime timestamp) that is calculated from the UNIX timestamp.

GETUTCDATE returns a universal (independent of time zone) date and time as either a %TimeStamp (ODBC timestamp) data type or a %PosixTime (encoded 64-bit signed integer) data type value. A %PosixTime value is calculated from the corresponding UNIX timestamp value. The %PosixTime encoding facilitates rapid timestamp comparisons and calculations. The %Library.PosixTime class provides a **UnixTimeToLogical()** method to convert a UNIX timestamp to a PosixTime timestamp, and a **LogicalToUnixTime()** method to convert a PosixTime timestamp to a UNIX timestamp. Neither of these methods perform timezone conversion.

You can also use the ObjectScript $ZTIMESTAMP special variable to return a universal (time zone independent) timestamp.

The ObjectScript $ZDATETIME function *dformat* -2 takes an InterSystems IRIS $HOROLOG date and returns a UNIX timestamp; $ZDATETIMEH *dformat* -2 takes a UNIX timestamp and returns an InterSystems IRIS %HOROLOG date. These ObjectScript functions convert local time to UTC time. **UNIX_TIMESTAMP** does not convert local time to UTC time.

Arguments

date-expression

An optional expression that is the name of a column, the result of another scalar function, or a date or timestamp literal. **UNIX_TIMESTAMP** does not convert from one timezone to another. If *date-expression* is omitted, defaults to the current UTC timestamp.

Examples

The following example returns a UTC UNIX timestamp. The first *select-item* takes the *date-expression* default, the second specifies an explicit UTC timestamp:

SQL

```
SELECT
    UNIX_TIMESTAMP() AS DefaultUTC,
    UNIX_TIMESTAMP(GETUTCDATE(3)) AS ExplicitUTC
```

The following example returns a local UNIX timestamp for the current local date and time, and a UTC UNIX timestamp for a UTC date and time value. The first *select-item* specifies the local CURRENT_TIMESTAMP, the second specifies $HOROLOG (local date and time), the third specifies the current UTC date and time:

SQL

```
SELECT
    UNIX_TIMESTAMP(CURRENT_TIMESTAMP(2)) AS CurrTSLocal,
    UNIX_TIMESTAMP($HOROLOG) AS HorologLocal,
    UNIX_TIMESTAMP(GETUTCDATE(3)) AS ExplicitUTC
```

The following example compares **UNIX_TIMESTAMP** (which does not convert local time) and **$ZDATETIME** (which does convert local time):

ObjectScript

```
SET unixutc=$ZDATETIME($HOROLOG,-2)
SET myquery = "SELECT UNIX_TIMESTAMP($HOROLOG) AS UnixLocal,? AS UnixUTC"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
 IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(unixutc)
DO rset.%Display()
```

See Also

- SQL concepts: Data Type, Date and Time Constructs

- SQL timestamp functions: CAST, CONVERT, GETDATE, GETUTCDATE, NOW, SYSDATE, TIMESTAMPADD, TIMESTAMPDIFF, TO_POSIXTIME, TO_TIMESTAMP

- ObjectScript: $ZDATETIME and $ZDATETIMEH functions, $HOROLOG special variable, $ZTIMESTAMP special variable

UPPER (SQL)

A case-transformation function that converts all lowercase letters in a string expression to uppercase letters.

Synopsis

```
UPPER(expression)
```

```
UPPER expression
```

Description

The **UPPER** function converts all alphabetic characters to uppercase letters. This is the inverse of the **LOWER** function. **UPPER** does not change numbers, punctuation, and leading or trailing blank spaces.

UPPER does not force a numeric to be interpreted as a string. InterSystems SQL removes leading and trailing zeros from numerics. A numeric specified as a string retains leading and trailing zeros.

This function can also be invoked from ObjectScript using the **UPPER()** method call:

```
$SYSTEM.SQL.Functions.UPPER(expression)
```

UPPER is a standard function for alphabetic case conversion, not for collation. For uppercase collation use %SQLUPPER, which provides superior collation of numerics, NULL values and empty strings.

Arguments

expression

A string expression, which can be the name of a column, a string literal, or the result of another function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR).

Examples

The following example returns all names, selecting those where the uppercase form of the name starts with "JO":

SQL

```
SELECT Name
FROM Sample.Person
WHERE UPPER(Name) %STARTSWITH UPPER('JO')
```

The following example returns all names in uppercase, selecting those where the name starts with "JO":

SQL

```
SELECT UPPER(Name) AS CapName
FROM Sample.Person
WHERE Name %STARTSWITH UPPER('JO')
```

The following example converts the lowercase Greek letter Delta to uppercase. This example uses the **UPPER** syntax that uses a space, rather than parentheses, to separate keyword from argument:

SQL

```
SELECT UPPER {fn CHAR(948)},{fn CHAR(948)}
FROM Sample.Person
```

See Also

- %SQLUPPER collation function

- **%STARTSWITH** predicate condition

- **LOWER** function

- **UCASE** function

- **Collation**

USER (SQL)

A function that returns the user name of the current user.

Synopsis

```
USER

{fn USER}
{fn USER()}
```

Description

USER takes no arguments and returns the user name (also referred to as the authorization ID) of the current user. The general function does not allow parentheses; the ODBC scalar function can specify or omit the empty parentheses.

A user name is defined with the **CREATE USER** command.

Typical uses for **USER** are in the **SELECT** statement select list or in the **WHERE** clause of a query. In designing a report, **USER** can be used to print the current user for whom the report is being produced.

Examples

The following example returns the current user name:

SQL

```
SELECT USER AS CurrentUser
```

The following example selects those records where the last name ($PIECE(Name,',',1) or the first name (without the middle initial) matches the current user name:

SQL

```
SELECT Name FROM Sample.Person
WHERE %SQLUPPER(USER)=%SQLUPPER($PIECE(Name,',',1))
OR %SQLUPPER(USER)=%SQLUPPER($PIECE($PIECE(Name,',',2),' ',1))
```

See Also

- CREATE USER, GRANT

WEEK (SQL)

A date function that returns the week of the year as an integer for a date expression.

Synopsis

```
{fn WEEK(date-expression)}
```

Description

WEEK takes a *date-expression*, and returns the number of weeks from the beginning of the year for that date.

By default, weeks are calculated using the $HOROLOG date (positive or negative integer number of days from Dec. 31, 1840). Therefore, weeks are counted from year to year, such that Week 1 is the days that complete the seven-day period begun by the last week of the previous year. A week always begins with a Sunday; therefore, the first Sunday of the calendar year marks the changing from Week 1 to Week 2. If the first Sunday of the year is January 1, then that Sunday is in Week 1; if the first Sunday of the year is later than January 1, then that Sunday is the first day of Week 2. For this reason, Week 1 is commonly less than seven days in length. You can determine the day of the week by using the DAYOFWEEK function. The total number of weeks in a year is commonly 53, and can be 54 in leap years.

InterSystems IRIS also supports the ISO 8601 standard for determining the week of the year. This standard is principally used in European countries. When InterSystems IRIS is configured for ISO 8601, **WEEK** begins counting a week with Monday, and assigns the week to the year that contains that week's Thursday. For example, Week 1 of 2004 ran from Monday 29 December 2003 to Sunday 4 January 2004, because this week's Thursday was 1 January 2004, which was the first Thursday of 2004. Week 1 of 2005 ran from Monday 3 January 2005 to Sunday 9 January 2005, because its Thursday was 6 January 2005, which was the first Thursday of 2005. The total number of weeks in a year is commonly 52, but can occasionally be 53. To activate ISO 8601 counting, `SET ^%SYS("sql","sys","week ISO8601")=1`.

The *date-expression* can be an InterSystems IRIS date integer, a $HOROLOG or $ZTIMESTAMP value, an ODBC format date string, or a timestamp.

A *date-expression* timestamp can be either data type %Library.PosixTime (an encoded 64-bit signed integer), or data type %Library.TimeStamp (yyyy-mm-dd hh:mm:ss.fff).

The time portion of the timestamp is not evaluated and can be omitted.

The same week information can be returned by using the DATEPART or DATENAME function.

This function can also be invoked from ObjectScript using the **WEEK()** method call:

```
$SYSTEM.SQL.Functions.WEEK(date-expression)
```

Date Validation

WEEK performs the following checks on input values. If a value fails a check, the null string is returned.

- A date string must be complete and properly formatted with the appropriate number of elements and digits for each element, and the appropriate separator character. Years must be specified as four digits.

- Date values must be within a valid range. Years: 0001 through 9999. Months: 1 through 12. Days: 1 through 31.

- The number of days in a month must match the month and year. For example, the date '02–29' is only valid if the specified year is a leap year.

- Date values less than 10 may include or omit a leading zero. Other non-canonical integer values are not permitted. Therefore, a Day value of '07' or '7' is valid, but '007', '7.0' or '7a' are not valid.

Arguments

date-expression

An expression that is the name of a column, the result of another scalar function, or a date or timestamp literal.

Examples

The following example returns the day of week and week of year for January 2, 2005 (which is a Sunday) and January 1, 2006 (which is a Sunday).

SQL

```
SELECT {fn DAYOFWEEK("2005-1-2")},{fn WEEK("2005-1-2")},
       {fn DAYOFWEEK("2006-1-1")},{fn WEEK("2006-1-1")}
```

The following examples return the number 9 because the date is the ninth week of the year 2004:

SQL

```
SELECT {fn WEEK('2004-02-25')} AS Wk_Date,
       {fn WEEK('2004-02-25 08:35:22')} AS Wk_Tstamp,
       {fn WEEK(59590)} AS Wk_DInt
```

The following example returns the number 54 because this particular date is in a leap year that began with Week 2 starting on the second day, as demonstrated by the example immediately following it:

SQL

```
SELECT {fn WEEK('2000-12-31')} AS Week
```

SQL

```
SELECT {fn WEEK('2000-01-01')}||{fn DAYNAME('2000-01-01')} AS WeekofDay1,
       {fn WEEK('2000-01-02')}||{fn DAYNAME('2000-01-02')} AS WeekofDay2
```

The following examples all return the current week:

SQL

```
SELECT {fn WEEK({fn NOW()})} AS Wk_Now,
       {fn WEEK(CURRENT_DATE)} AS Wk_CurrD,
       {fn WEEK(CURRENT_TIMESTAMP)} AS Wk_CurrTS,
       {fn WEEK($HOROLOG)} AS Wk_Horolog,
       {fn WEEK($ZTIMESTAMP)} AS Wk_ZTS
```

The following Embedded SQL example shows the InterSystems IRIS default week of the year and the week of the year with the ISO 8601 standard applied:

ObjectScript

```
TestISO
  SET def=$DATA(^%SYS("sql","sys","week ISO8601"))
  IF def=0 {SET ^%SYS("sql","sys","week ISO8601")=0}
  ELSE {SET isoval=^%SYS("sql","sys","week ISO8601")}
     IF isoval=1 {GOTO UnsetISO }
     ELSE {SET isoval=0 GOTO WeekOfYear }
UnsetISO
  SET ^%SYS("sql","sys","week ISO8601")=0
WeekOfYear
  &sql(SELECT {fn WEEK($HOROLOG)} INTO :a)
  WRITE "For Today:",!
  WRITE "default week of year is ",a,!
  SET ^%SYS("sql","sys","week ISO8601")=1
  &sql(SELECT {fn WEEK($HOROLOG)} INTO :b)
  WRITE "ISO8601 week of year is ",b,!
ResetISO
  SET ^%SYS("sql","sys","week ISO8601")=isoval
```

See Also

- SQL functions: DATENAME, DATEPART, DAYOFWEEK, MONTH, QUARTER, TO_DATE, YEAR

- ObjectScript special variables: $HOROLOG, $ZTIMESTAMP

XMLCONCAT (SQL)

A function that concatenates XML elements.

Synopsis

```
XMLCONCAT(XmlElement1,XmlElement2[,...])
```

Description

The **XMLCONCAT** function returns the values from several **XMLELEMENT** functions as a single string. **XMLCONCAT** can be used in a SELECT query or subquery that references either a table or a view. **XMLCONCAT** can appear in a **SELECT** list alongside ordinary field values.

Arguments

XmlElement

An XMLELEMENT function. Specify two or more *XmlElement* to concatenate.

Examples

The following query concatenates the values from two **XMLELEMENT** functions:

SQL

```
SELECT Name,XMLCONCAT(XMLELEMENT("Para",Name),
                      XMLELEMENT("Para",Home_City)) AS ExportString
    FROM Sample.Person
```

A sample row of the data returned would appear as follows:

```
ExportString
<Para>Emerson,Molly N.</Para><Para>Boston</Para>
```

The following query nests an **XMLCONCAT** within an **XMLELEMENT** function:

SQL

```
SELECT XMLELEMENT("Item",Name,
         XMLCONCAT(
             XMLELEMENT("Para",Home_City,' ',Home_State),
             XMLELEMENT("Para",'is residence')))
       AS ExportString
FROM Sample.Person
```

A sample row of the data returned would appear as follows:

```
ExportString
<Item>Emerson,Molly N.<Para>Boston MA</Para><Para>is residence</Para></Item>
```

See Also

SELECT statement

XMLAGG function

XMLELEMENT function

XMLELEMENT (SQL)

A function that formats an XML markup tag to enclose one or more expression values.

Synopsis

```
XMLELEMENT([NAME] tag,expression[,expression])

XMLELEMENT([NAME] tag,XMLATTRIBUTES(expression [AS alias]),expression[,expression])
```

Description

The **XMLELEMENT** function returns the values of *expression* tagged with the XML (or HTML) markup start-tag and end-tag specified in *tag*. For example, XMLELEMENT(NAME "Para",Home_City) returns values such as the following: <Para>Chicago</Para>. **XMLELEMENT** cannot be used to generate an empty-element tag.

XMLELEMENT can be used in a SELECT query or subquery that references either a table or a view. **XMLELEMENT** can appear in a **SELECT** list alongside ordinary field values.

The *tag* argument uses double quotes to enclose a literal string. In nearly all other contexts, InterSystems SQL uses single quotes to enclose a literal string; it uses double quotes to specify a delimited identifier. Therefore, delimited identifier support must be enabled to use this feature; delimited identifiers are enabled by default.

When SQL code is specified as a string delimited by double quotes, such as in a Dynamic SQL **%Prepare()** method, you must escape the *tag* double quotes by specifying two double quotes, as follows:

ObjectScript

```
SET myquery = "SELECT XMLELEMENT(""Para"",Name) FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
```

Commonly, *expression* is the name of a field, (or an expression containing one or more field names) in the multiple rows returned by a query. An *expression* can be a field of any type. The specified *expression* value is returned enclosed by a start tag and an end tag, as shown in the following format:

```
<tag>value</tag>
```

If the value to be tagged is either the empty string (") value or a NULL, the following is returned:

```
<tag></tag>
```

If the *expression* contains multiple comma-separated elements, the results are concatenated, as shown in the following format:

```
<tag>value1value2</tag>
```

If the *expression* is a data stream field, the stream value is escaped within the resulting XML value using <![CDATA[...]]>:

```
<tag><![CDATA[value]]></tag>
```

XMLELEMENT functions can be nested. **XMLELEMENT** and **XMLFOREST** functions may be nested in any combination. **XMLELEMENT** functions can be concatenated using XMLCONCAT. However, **XMLELEMENT** does not do XML type resolution of entire expressions. For example, **XMLELEMENT** cannot perform character conversion within a clause of a **CASE** statement (see example below).

XMLATTRIBUTES Function

The **XMLATTRIBUTES** function can only be used within an **XMLELEMENT** function. If an element of *expression* is an **XMLATTRIBUTES** function, the specified expression becomes an attribute of the tag, as shown in the following format:

```
<tag ID='63' >value</tag>
```

You can only specify one **XMLATTRIBUTES** function within an **XMLELEMENT** function. By convention it is the first *expression* element, though it can be any element in *expression*. InterSystems IRIS encloses attribute values with single quotes and inserts a space between the attribute value and the closing angle bracket (>) for the tag.

XMLELEMENT and XMLFOREST Compared

- **XMLELEMENT** concatenates the values of its *expression* list within a single tag. **XMLFOREST** assigns a separate tag for each *expression* item.

- **XMLELEMENT** requires that you specify a tag value. **XMLFOREST** allows you to either take default tag values or specify individual tag values.

- **XMLELEMENT** allows you to specify a tag attribute using **XMLATTRIBUTES**. **XMLFOREST** does not allow you to specify a tag attribute.

- **XMLELEMENT** returns a tag string for NULL. **XMLFOREST** does not return a tag string for NULL.

Punctuation Character Values

If a data value contains a punctuation character that XML/HTML might interpret as a tag or other coding, **XMLELEMENT** and **XMLFOREST** convert this character to the corresponding encoded form:

ampersand (&) becomes &

apostrophe (') becomes '

quotation mark (") becomes "

open angle bracket (<) becomes <

close angle bracket (>) becomes >

To represent an apostrophe in a supplied text string, specify two apostrophes, as in the following example: `'can''t'`. Doubling apostrophes is not necessary for column data.

Arguments

NAME tag

The name of an XML markup tag. The NAME keyword is optional. This argument has three syntactical forms: `NAME "tag"`, `"tag"`, and `NAME`. The first two are functionally identical. If specified, *tag* must be enclosed in double quotes. The case of letters in *tag* is preserved.

XMLELEMENT performs no validation of *tag* values. However, the XML standard requires that a valid *tag* name cannot contain any of the characters ! "#$%&'()*+,/;<=>?@[\]^`{|}~, nor a space character, and cannot begin with "-", ".", or a numeric digit.

If you specify the NAME keyword without a *tag* value, InterSystems IRIS supplies the default tag value: <Name> ... </Name>. The NAME keyword is not case-sensitive; the resulting tag is initial capitalized.

expression

Any valid expression. Usually the name of a column that contains the data values to be tagged. You can specify a comma-separated list of columns or other expressions, all of which will be enclosed within the same *tag*. The first comma-separated element can be an **XMLATTRIBUTES** function. Only one **XMLATTRIBUTES** element can be specified.

Examples

The following example returns each person's Name field value in Sample.Person as ordinary data and as xml tagged data:

SQL

```
SELECT Name,
       XMLELEMENT("Para",Name) AS ExportName
    FROM Sample.Person
```

A sample row of the data returned would appear as follows:

```
Name                    ExportName
Emerson,Molly N.    <Para>Emerson,Molly N.</Para>
```

The following example returns every distinct Home_City and Home_State pair value in Sample.Person as xml tagged data with the tag <Address> ... </Address>. A blank space *expression* is specified to prevent concatenation of the city name and the state name:

```
SELECT DISTINCT
    XMLELEMENT(NAME "Address",Home_City,' ',Home_State) AS CityState
    FROM Sample.Person
    ORDER BY Home_City
```

Note that in the above example the optional NAME keyword is supplied. In the next example, the NAME keyword is provided without the *tag* value:

SQL

```
SELECT DISTINCT
    XMLELEMENT(NAME,Home_City,' ',Home_State) AS CityState
    FROM Sample.Person
    ORDER BY Home_City
```

In this case the same data is returned, but is tagged with the default tag: <Name> ... </Name>.

The following example returns character stream data:

SQL

```
SELECT XMLELEMENT("Para",Name) AS XMLNotes,XMLELEMENT("Para",Notes) AS XMLText
    FROM Sample.Employee
```

A sample row of the data returned would appear as follows:

```
XMLName                         XMLText
<Para>Emerson,Molly N.</Para>    <Para><![CDATA[Molly worked at DynaMatix Holdings Inc. as a Marketing
 Manager]]></Para>
```

The following example shows that **XMLELEMENT** functions can be nested:

SQL

```
SELECT XMLELEMENT("Para",Home_State,
                  XMLELEMENT("Emphasis",Name),Age)
FROM Sample.Person
```

A sample row of the data returned would appear as follows:

```
<Para>CA<Emphasis>Emerson,Molly N.</Emphasis>24</Para>
```

The following example shows **XMLELEMENT** functions using a subquery value:

SQL

```
SELECT XMLELEMENT("Para",Name,DOB, XMLELEMENT("Emphasis",%ID),Age,
    (SELECT XMLELEMENT("NameSub",Name) FROM Sample.Person WHERE %ID=2)) AS ExportName
FROM Sample.Person WHERE %ID=1
```

A sample row of the data returned would appear as follows:

```
<Para>Zucherro,Rob F.38405<Emphasis>1</Emphasis>71<NameSub>Quixote,Mark N.</NameSub></Para>
```

The following example shows that **XMLELEMENT** *can not* tag a value within a CASE statement clause:

SQL

```
SELECT XMLELEMENT("Para",Home_State,
    XMLELEMENT("Para",Name),
    CASE WHEN Age < 21 THEN NULL
      ELSE XMLELEMENT("Para",Age) END )
FROM Sample.Person
```

A sample row of the data returned would appear as follows:

```
<Para>CA<Para>Emerson,Molly N.</Para>&lt;Para&gt;24&lt;/Para&gt;</Para>
```

The following query returns the Name field values in Sample.Person as XML-tagged data in a tag that uses the ID field as a tag attribute:

SQL

```
SELECT XMLELEMENT("Para",XMLATTRIBUTES(%ID),Name) AS ExportName
    FROM Sample.Person
```

A sample row of the data returned would appear as follows:

```
ExportName
<Para ID='101' >Emerson,Molly N.</Para>
```

You can specify an alias for an attribute, as shown in the following example:

SQL

```
SELECT XMLELEMENT("Para",XMLATTRIBUTES(%ID AS ItemKey),Name)
    FROM Sample.Person
```

A sample row of the data returned would appear as follows:

```
<Para ItemKey='101' >Emerson,Molly N.</Para>
```

See Also

XMLAGG function

XMLCONCAT function

XMLFOREST function

SELECT statement

XMLFOREST (SQL)

A function that formats multiple XML markup tags to enclose expression values.

Synopsis

```
XMLFOREST(expression [AS tag][,expression [AS tag]])
```

Description

The **XMLFOREST** function returns the values of each *expression* tagged with its own XML markup start-tag and end-tag, as specified in *tag*. For example, XMLFOREST(Home_City AS City,Home_State AS State) returns values such as the following: <City>Chicago</City><State>IL</State>. **XMLFOREST** cannot be used to generate an empty-element tag.

XMLFOREST can be used in a SELECT query or subquery that references either a table or a view. **XMLFOREST** can appear in a **SELECT** list alongside ordinary column values.

The specified *expression* value is returned enclosed by a start tag and an end tag, as shown in the following format:

```
<tag>value</tag>
```

Commonly, *expression* is the name of a column, or an expression containing one or more column names. An *expression* can be a field of any type, including a data stream field. **XMLFOREST** tags each *expression* as follows:

- If AS *tag* is specified, **XMLFOREST** tags the resulting values with the specified tag. The *tag* value is case-sensitive.

- If AS *tag* is omitted, and *expression* is a column name, **XMLFOREST** tags the resulting values with the column name. Column name default tags are always uppercase.

- If *expression* is not a column name (for example, an aggregate function, a literal, or a concatenation of two columns) the AS *tag* clause is required.

- If *expression* is a stream field, the stream value is escaped within the resulting XML value using <![CDATA[...]]>:

  ```
  <tag><![CDATA[value]]></tag>
  ```

XMLFOREST provides a separate tag for each item in a comma-separated list. **XMLELEMENT** concatenates all of the items in a comma-separated list within a single tag.

XMLFOREST functions can be nested. Any combination of nested **XMLFOREST** and **XMLELEMENT** functions is permitted. **XMLFOREST** functions can be concatenated using XMLCONCAT.

NULL Values

The **XMLFOREST** function only returns a tag for actual data values. It does not return a tag when the *expression* value is NULL. For example:

SQL

```
INSERT INTO Sample.Xmltest (f1,f2,f3) values (NULL,'Row 1',NULL)
```

SQL

```
SELECT XMLFOREST(f1,f2,f3) from Sample.Xmltest
```

returns: <F2>Row 1</F2>.

The empty string (") is considered a data value for a string data type field. If the f3 value to be tagged is the empty string ("), **XMLFOREST** returns:

```
<F3></F3>
```

XMLFOREST differs from **XMLELEMENT** in the handling of NULL. **XMLELEMENT** always returns a tag value, even when the field value is NULL. **XMLELEMENT** therefore does not distinguish between a NULL or an empty string. Both are represented as `<tag></tag>`.

Punctuation Character Values

If a data value contains a punctuation character that XML/HTML might interpret as a tag or other coding, **XMLFOREST** and **XMLELEMENT** convert this character to the corresponding encoded form:

ampersand (&) becomes `&`

apostrophe (') becomes `'`

quotation mark (") becomes `"`

open angle bracket (<) becomes `<`

close angle bracket (>) becomes `>`

To represent an apostrophe in a supplied text string, specify two apostrophes, as in the following example: `'can''t'`. Doubling apostrophes is not necessary for column data.

Arguments

expression

Any valid expression. Usually the name of a column that contains the data values to be tagged. When specified as a comma-separated list, each expression in the list will be enclosed in its own XML markup tag.

AS tag

An optional argument that specifies the name of an XML markup tag. The AS keyword is mandatory if *tag* is specified. The case of letters in *tag* is preserved.

Enclosing *tag* with double quotes is optional. If you omit the double quotes, *tag* must follow XML naming standards. Enclosing *tag* with double quotes removes these naming restrictions.

XMLFOREST enforces XML naming standards for a valid *tag* name. It cannot contain any of the characters `!"#$%&'()*+,/;<=>?@[\]^`{|}~`, nor a space character, and cannot begin with "-", ".", or a numeric digit.

If you specify an *expression* without the AS *tag* clause, the tag value is the name of the *expression* column (in capital letters): `<HOME_CITY>Chicago</HOME_CITY>`.

Examples

The following query returns the Name column values in Sample.Person as ordinary data and as xml tagged data:

SQL

```
SELECT Name,XMLFOREST(Name) AS ExportName
    FROM Sample.Person
```

A sample row of the data returned would appear as follows. Here the tag defaults to the name of the column:

```
Name                    ExportName
Emerson,Molly N.    <NAME>Emerson,Molly N.</NAME>
```

The following example specifies multiple columns:

SQL

```
SELECT XMLFOREST(Home_City,
                 Home_State AS Home_State,
                 AVG(Age) AS AvAge) AS ExportData
FROM Sample.Person
```

The Home_City field specifies no tag; the tag is generated from the column name in all capital letters: <HOME_CITY>. The Home_State field's AS clause is optional. It is specified here because specifying the tag name allows you to control the case of the tag: <Home_State>, rather than <HOME_STATE>. The AVG(Age) AS clause is mandatory, because the value is an aggregate, not a column value, and thus has no column name. A sample row of the data returned would appear as follows.

```
ExportData
<HOME_CITY>Chicago</HOME_CITY><Home_State>IL</Home_State>
<AvAge>48.0198019801980198</AvAge>
```

The following example returns character stream data:

SQL

```
SELECT XMLFOREST(name AS Para,Notes AS Para) AS XMLJobHistory
    FROM Sample.Employee
```

A sample row of the data returned would appear as follows:

```
XMLJobHistory
<Para>Emerson,Molly N.</Para><Para><![CDATA[Molly worked at DynaMatix Holdings Inc. as a Marketing
Manager]]></Para>
```

The following example shows **XMLFOREST** functions using a subquery value:

SQL

```
SELECT XMLFOREST(Name,DOB,Age,
    (SELECT XMLFOREST(Name,DOB) FROM Sample.Person WHERE %ID=2) AS ExportName)
FROM Sample.Person where %ID=1
```

A sample row of the data returned would appear as follows:

```
<NAME>Zahn,Rob F.</NAME><DOB>38405</DOB><AGE>71</AGE><ExportName><NAME>Quinn,Mark
N.</NAME><DOB>30999</DOB></ExportName>
```

See Also

XMLAGG function

XMLELEMENT function

XMLCONCAT function

SELECT statement

YEAR (SQL)

A date function that returns the year for a date expression.

Synopsis

```
YEAR(date-expression)
```

```
{fn YEAR(date-expression)}
```

Description

YEAR takes as input an InterSystems IRIS date integer ($HOROLOG date), an ODBC format date string, or a timestamp. **YEAR** returns the corresponding year as an integer.

A *date-expression* timestamp can be either data type %Library.PosixTime (an encoded 64-bit signed integer), or data type %Library.TimeStamp (yyyy-mm-dd hh:mm:ss.fff).

The year (yyyy) portion should be a four-digit integer in the range 0001 through 9999. Leading zeros are optional on input. Leading zeros are suppressed on output. Two digit years are *not* expanded to four digits.

The date portion of *date-expression* is validated and must include a month within the range 1 through 12 and a valid day value for the specified month and year. Otherwise, an SQLCODE -400 error <ILLEGAL VALUE> is generated.

The time portion of *date-expression* is validated if present, but can be omitted.

Note: For compatibility with InterSystems IRIS internal representation of dates, it is *strongly* recommended that all year values be expressed as four-digit integers within the range of 0001 through 9999.

The **TO_DATE** and **TO_CHAR** SQL functions support "Julian dates," which can be used to represent years before 0001. ObjectScript provides method calls that support such Julian dates.

The year format default is four-digit years. To change this year display default, use the SET OPTION command with the YEAR_OPTION option.

The elements of a datetime string can be returned using the following SQL scalar functions: **YEAR**, **MONTH**, **DAY**, **DAYOFMONTH**, **HOUR**, **MINUTE**, **SECOND**. The same elements can be returned by using the DATEPART or DATENAME function.

This function can also be invoked from ObjectScript using the **YEAR()** method call:

```
$SYSTEM.SQL.Functions.YEAR(date-expression)
```

Arguments

date-expression

An expression that evaluates to either an InterSystems IRIS date integer, an ODBC date string, or a timestamp. This expression can be the name of a column, the result of another scalar function, or a date or timestamp literal.

Examples

The following examples return the integer 2018:

SQL

```
SELECT YEAR('2018-02-22 12:45:37') AS ODBCDate_Year
```

SQL

```
SELECT {fn YEAR(64701)} AS HorologDate_Year
```

The following example returns the current year:

SQL

```
SELECT YEAR(GETDATE()) AS Year_Now
```

The following example returns the current year from two functions. The **CURRENT_DATE** function returns data type DATE; the **NOW** function returns data type TIMESTAMP. **YEAR** returns a four-digit year integer for both input data types:

SQL

```
SELECT {fn YEAR(CURRENT_DATE)}, {fn YEAR({fn NOW()})}
```

See Also

- SQL functions: DATENAME, DATEPART, DAYOFYEAR, QUARTER, WEEK, TO_DATE

- ObjectScript function: $ZDATE

SQL Unary Operators

- (Negative)

A unary operator that returns an expression as a negative, numeric value.

Synopsis

```
-expression
```

Description

Unary operators perform an operation on only one expression of any of the data types of the numeric data type category.

– (Negative) is an InterSystems SQL extension.

Arguments

expression

A numeric expression.

Examples

The following example returns three numeric fields: the Age column from Sample.Person; the – (Negative) value of the average of Age; and the Age minus the average age:

SQL

```
SELECT Age,
       -(AVG(age)) AS NegAvg,
       Age-AVG(Age) AS AgeRelAvg
FROM Sample.Person
```

See Also

+ (Positive)

+ (Positive)

A unary operator that returns an expression as a positive, numeric value.

Synopsis

```
+expression
```

Description

Unary operators perform an operation on only one expression. This expression can be any of the data types of the numeric data type category.

+ (Positive) is an InterSystems SQL extension.

Arguments

expression

A numeric expression.

See Also

- (Negative)

SQL Reference Material

Data Types (SQL)

Specifies the kind of data that an SQL column can contain.

Data Types in InterSystems SQL

A *data type* specifies the kind of value that a table column can hold. In InterSystems SQL, you specify the data type when defining a field with **CREATE TABLE** or **ALTER TABLE**. You can define either a Data Definition Language (DDL) data type or an InterSystems IRIS data type class. For example:

DDL Using SQL Data Types

```
CREATE TABLE Employees (
    FirstName VARCHAR(30),
    LastName VARCHAR(30),
    StartDate TIMESTAMP)
```

DDL Using InterSystems IRIS Data Types

```
CREATE TABLE Employees (
    FirstName %String(MAXLEN=30),
    LastName %String(MAXLEN=30),
    StartDate %TimeStamp)
```

View Data Type Mappings to InterSystems IRIS

Each DDL data type maps to an equivalent InterSystems IRIS data type. To view the standard mappings for your system:

1. From the Management Portal, select **System Administration**.

2. Under **Configuration** and then **SQL and Object Settings**, click **System DDL Mappings**.

The **System-defined DDL Mappings** page shows a table with these columns:

- `Name` — The name of a DDL data type that you can specify. DDL data type names are case insensitive.

- `Datatype` — The name of the InterSystems IRIS class data type that the DDL data type maps to. Class names are case sensitive.

From the **System-defined DDL Mappings** page, you can modify and delete existing data types. This table explains the mappings, including the literal data type parameters such as `%1` and function parameters such as **$$maxval^%apiSQL(%1,%2)**. For more details about these parameters, see Data Type Mapping Parameters.

DDL Data Type	Corresponding InterSystems IRIS Data Type Class
BIGINT	%Library.BigInt If a BIGINT column can contain both NULLs and extremely small negative numbers, you might need to redefine the index null marker to support standard index collation. For more details, see Indexing a NULL.
BIGINT(%1)	%Library.BigInt The %1 parameter is ignored and is provided for MySQL compatibility. This data type is equivalent to BIGINT.
BINARY	%Library.Binary(MAXLEN=1)

DDL Data Type	Corresponding InterSystems IRIS Data Type Class
BINARY VARYING	%Library.Binary(MAXLEN=1)
BINARY VARYING(%1)	%Library.Binary(MAXLEN=%1) %1 sets the maximum length of the data type.
BINARY(%1)	%Library.Binary(MAXLEN=%1) %1 sets the maximum length of the data type.
BIT	%Library.Boolean For more details on this data type, see BIT Data Type.
BLOB	%Stream.GlobalBinary
CHAR	%Library.String(MAXLEN=1)
CHAR VARYING	%Library.String(MAXLEN=1)
CHAR VARYING(%1)	%Library.String(MAXLEN=%1) %1 sets the maximum length of the data type.
CHAR(%1)	%Library.String(MAXLEN=%1) %1 sets the maximum length of the data type.
CHARACTER	%Library.Binary(MAXLEN=1)
CHARACTER VARYING	%Library.String(MAXLEN=1)
CHARACTER VARYING(%1)	%Library.String(MAXLEN=%1) %1 sets the maximum length of the data type.
CHARACTER(%1)	%Library.String(MAXLEN=%1) %1 sets the maximum length of the data type.
CLOB	%Stream.GlobalCharacter
DATE	%Library.Date
DATETIME	%Library.DateTime
DATETIME2	%Library.DateTime
DEC	%Library.Numeric(MAXVAL=999999999999999, MIN-VAL=-999999999999999, SCALE=0)

DDL Data Type	Corresponding InterSystems IRIS Data Type Class
DEC(%1)	%Library.Numeric(MAXVAL=<|'$$max-val^%apiSQL(%1,0)'\|>,MINVAL=<\|'$$min-val^%apiSQL(%1,0)'|>,SCALE=0) This data type uses function parameters to set MINVAL and MAXVAL based on the input precision parameter (%1) with the scale set to 0. For more details on these parameters, see Precision and Scale. Example: DEC(4) maps to%Library.Numeric(MAXVAL=9999,MINVAL=–9999,SCALE=0)
DEC(%1,%2)	%Library.Numeric (MAXVAL=<|'$$max-val^%apiSQL(%1,%2)'\|>, MINVAL=<\|'$$min-val^%apiSQL(%1,%2)'|>, SCALE=%2) This data type uses function parameters to set MINVAL, MAXVAL, and SCALE based on the input precision (%1) and scale (%2) parameters. For more details on these parameters, see Precision and Scale. Example: DEC(8,4) maps to %Library.Numeric(MAXVAL=9999.9999,MINVAL=-9999.9999,SCALE=4)
DECIMAL	%Library.Numeric(MAXVAL=999999999999999, MINVAL=-999999999999999, SCALE=0)
DECIMAL(%1)	%Library.Numeric(MAXVAL=<|'$$max-val^%apiSQL(%1,0)'\|>,MINVAL=<\|'$$min-val^%apiSQL(%1,0)'|>,SCALE=0) This data type uses function parameters to set MINVAL and MAXVAL based on the input precision parameter (%1) with the scale set to 0. For more details on these parameters, see Precision and Scale. This data type is a 64-bit signed integer. Example: DECIMAL(6) maps to %Library.Numeric(MAXVAL=999999,MINVAL=-999999,SCALE=0)
DECIMAL(%1,%2)	%Library.Numeric (MAXVAL=<|'$$max-val^%apiSQL(%1,%2)'\|>, MINVAL=<\|'$$min-val^%apiSQL(%1,%2)'|>, SCALE=%2) This data type uses function parameters to set MINVAL, MAXVAL, and SCALE based on the input precision (%1) and scale (%2) parameters. For more details on these parameters, see Precision and Scale. Example: DECIMAL(8,4) maps to%Library.Numeric(MAXVAL=9999.9999,MINVAL=-9999.9999,SCALE=4)

DDL Data Type	Corresponding InterSystems IRIS Data Type Class
DOUBLE	%Library.Double This is the IEEE floating point standard. An SQL column with this data type returns a default precision of 20. For further details (including important max/min value limits), see the $DOUBLE function.
DOUBLE PRECISION	%Library.Double This is the IEEE floating point standard. An SQL column with this data type returns a default precision of 20. For further details (including important max/min value limits), see the $DOUBLE function.
FLOAT	%Library.Double This is the IEEE floating point standard. An SQL column with this data type returns a default precision of 20.
FLOAT(%1)	%Library.Double This is the IEEE floating point standard. An SQL column with this data type returns a default precision of 20.
IMAGE	%Stream.GlobalBinary
INT	%Library.Integer (MAXVAL=2147483647, MINVAL=-2147483648)
INT(%1)	%Library.Integer (MAXVAL=2147483647, MINVAL=-2147483648) The %1 parameter is ignored and is provided for MySQL compatibility. This data type is equivalent to INT.
INTEGER	%Library.Integer (MAXVAL=2147483647, MINVAL=-2147483648)
LONG	%Stream.GlobalCharacter
LONG BINARY	%Stream.GlobalBinary
LONG RAW	%Stream.GlobalBinary
LONG VARCHAR	%Stream.GlobalCharacter
LONG VARCHAR(%1)	%Stream.GlobalCharacter The %1 parameter is ignored and is provided for MySQL compatibility.

DDL Data Type	Corresponding InterSystems IRIS Data Type Class
LONGTEXT	%Stream.GlobalCharacter The %1 parameter is ignored and is provided for MySQL compatibility.
LONGVARBINARY	%Stream.GlobalBinary
LONGVARBINARY(%1)	%Stream.GlobalBinary The %1 parameter is ignored and is provided for MySQL compatibility.
LONGVARCHAR	%Stream.GlobalCharacter
LONGVARCHAR(%1)	%Stream.GlobalCharacter
MEDIUMINT	%Library.Integer(MAXVAL=8388607,MINVAL=-8388608) This data type is provided for MySQL compatibility.
MEDIUMINT(%1)	%Library.Integer(MAXVAL=8388607,MINVAL=-8388608) The %1 parameter is ignored and is provided for MySQL compatibility.
MEDIUMTEXT	%Stream.GlobalCharacter
MONEY	%Library.Currency
NATIONAL CHAR	%Library.String(MAXLEN=1)
NATIONAL CHAR VARYING	%Library.String(MAXLEN=1)
NATIONAL CHAR VARYING(%1)	%Library.String(MAXLEN=%1) %1 sets the maximum length of the data type.
NATIONAL CHAR(%1)	%Library.String(MAXLEN=%1) %1 sets the maximum length of the data type.
NATIONAL CHARACTER	%Library.String(MAXLEN=1)
NATIONAL CHARACTER VARYING	%Library.String(MAXLEN=1)
NATIONAL CHARACTER VARYING(%1)	%Library.String(MAXLEN=%1) %1 sets the maximum length of the data type.
NATIONAL CHARACTER(%1)	%Library.String(MAXLEN=%1) %1 sets the maximum length of the data type.
NATIONAL VARCHAR	%Library.String(MAXLEN=1)
NATIONAL VARCHAR(%1)	%Library.String(MAXLEN=%1) %1 sets the maximum length of the data type.

DDL Data Type	Corresponding InterSystems IRIS Data Type Class
NCHAR	%Library.String(MAXLEN=1)
NCHAR(%1)	%Library.String(MAXLEN=%1) %1 sets the maximum length of the data type.
NTEXT	%Stream.GlobalCharacter
NUMBER	%Library.Numeric(SCALE=0) This data type is a 64-bit signed integer.
NUMBER(%1)	%Library.Numeric(MAXVAL=<|'$$max-val^%apiSQL(%1)'\|>,MINVAL=<\|'$$min-val^%apiSQL(%1)'|>,SCALE=0) This data type uses function parameters to set MIN-VAL and MAXVAL based on the input precision parameter (%1) with the scale set to 0. This data type is a 64-bit signed integer. Example: NUMBER(6) maps to %Library.Numeric(MAXVAL=999999,MINVAL=-999999,SCALE=0)
NUMBER(%1,%2)	%Library.Numeric (MAXVAL=<|'$$max-val^%apiSQL(%1,%2)'\|>, MINVAL=<\|'$$min-val^%apiSQL(%1,%2)'|>, SCALE=%2) This data type uses function parameters to set MIN-VAL, MAXVAL, and SCALE based on the input precision (%1) and scale (%2) parameters. Example: NUMBER(8,4) maps to%Library.Numeric(MAXVAL=9999.9999,MINVAL=-9999.9999,SCALE=4)
NUMERIC	%Library.Numeric(MAXVAL=999999999999999, MINVAL=-999999999999999, SCALE=0)
NUMERIC(%1)	%Library.Numeric(MAXVAL=<|'$$max-val^%apiSQL(%1,0)'\|>,MINVAL=<\|'$$min-val^%apiSQL(%1,0)'|>,SCALE=0) This data type uses function parameters to set MIN-VAL and MAXVAL based on the input precision parameter (%1) with the scale set to 0. This data type is a 64-bit signed integer. Example: NUMERIC(6) maps to %Library.Numeric(MAXVAL=999999,MINVAL=-999999,SCALE=0)

DDL Data Type	Corresponding InterSystems IRIS Data Type Class
NUMERIC(%1,%2)	%Library.Numeric (MAXVAL=<|'$$max-val^%apiSQL(%1,%2)'\|>, MINVAL=<\|'$$min-val^%apiSQL(%1,%2)'|>, SCALE=%2)
	This data type uses function parameters to set MINVAL, MAXVAL, and SCALE based on the input precision (%1) and scale (%2) parameters.
	Example: NUMERIC(8,4) maps to%Library.Numeric(MAXVAL=9999.9999,MINVAL=-9999.9999,SCALE=4)
NVARCHAR	%Library.String(MAXLEN=1)
NVARCHAR(%1)	%Library.String(MAXLEN=%1)
	%1 sets the maximum length of the data type.
NVARCHAR(%1,%2)	%Library.String(MAXLEN=%1)
NVARCHAR(MAX)	%Stream.GlobalCharacter
	This data type is equivalent to LONGVARCHAR and is provided for TSQL compatibility.
POSIXTIME	%Library.PosixTime
RAW(%1)	%Library.Binary(MAXLEN=%1)
	%1 sets the maximum length of the data type.
REAL	%Library.Double
	This is the IEEE floating point standard. An SQL column with this data type returns a default precision of 20.
ROWVERSION	%Library.RowVersion
	This data type is a system-assigned sequential integer. See ROWVERSION Data Type for details.
SERIAL	%Library.Counter
	This data type is system-generated
SMALLDATETIME	%Library.DateTime(MINVAL="1900-01-01 00:00:00",MAXVAL="2079-06-06 23:59:59")
SMALLINT	%Library.SmallInt
SMALLINT(%1)	%Library.SmallInt
	The %1 parameter is ignored and is provided for MySQL compatibility. This data type is equivalent to SMALLINT.

DDL Data Type	Corresponding InterSystems IRIS Data Type Class
SMALLMONEY	%Library.Currency
SYSNAME	%Library.String(MAXLEN=128)
TEXT	%Stream.GlobalCharacter
TIME	%Library.Time
TIME(%1)	%Library.Time(PRECISION=%1) PRECISION is the number of fractional second digits, an integer value in the range 0 through 9.
TIMESTAMP	%Library.PosixTime
TIMESTAMP2	%Library.TimeStamp
TINYINT	%Library.TinyInt
TINYINT(%1)	%Library.TinyInt The %1 parameter is ignored and is provided for MySQL compatibility. This data type is equivalent to TINYINT.
UNIQUEIDENTIFIER	%Library.UniqueIdentifier
VARBINARY	%Library.Binary(MAXLEN=1)
VARBINARY(%1)	%Library.Binary(MAXLEN=%1) %1 sets the maximum length of the data type.
VARCHAR	%Library.String(MAXLEN=1)
VARCHAR(%1)	%Library.String(MAXLEN=%1) %1 sets the maximum length of the data type.
VARCHAR(%1,%2)	%Library.String(MAXLEN=%1) %1 sets the maximum length of the data type.
VARCHAR(MAX)	%Stream.GlobalCharacter This data type is equivalent to LONGVARCHAR and is provided for TSQL compatibility.
VARCHAR2(%1)	%Library.String(MAXLEN=%1) %1 sets the maximum length of the data type.

Data Type Mapping Parameters

The **System-defined DDL Mappings** table often includes multiple entries for the same data type to show the different parameters you can specify for that data type. The mapping table also shows parameter default values. Data type classes commonly provide additional parameters to define allowed data values than the DDL data types. You can specify either literal parameters or function parameters. You can also define additional data type class parameters.

Literal Parameters

Literal parameters are identified in the DDL data type and the InterSystems IRIS data type in the format %*n*, where *n* is the number of the data type argument. For example, `VARCHAR(%1)` maps to `%String(MAXLEN=%1)`

Common literal parameters include maximum string length, minimum and maximum values, and precision and scale values.

Maximum Length

In data type classes, the MAXLEN parameter specifies the maximum length of string data types. DDL types often define these values in a corresponding unnamed parameter.

In this field definition, the data type is a string with a maximum length of 64 characters.

DDL Using SQL Data Type

```
ProductName VARCHAR(64)
```

DDL Using InterSystems IRIS Data Type

```
ProductName %String(MAXLEN=64)
```

When specifying this parameter, keep these points in mind:

- A field with no MAXLEN value can take a value of any length, up to the maximum string length. To define a string field of maximum length, specify VARCHAR(''), which create a property with data type %Library.String(MAXLEN=''). VARCHAR() creates a property with data type %Library.String(MAXLEN=1). To define a binary field with no MAXLEN value, specify VARBINARY(''), which create a property with data type %Library.Binary(MAXLEN=''). VARBINARY() creates a property with data type %Library.Binary(MAXLEN=1).

- Large MAXLEN: ODBC applications may be affected by an overly large MAXLEN value. ODBC applications try to make decisions about the size of a field needed based on metadata from the server, so the application may allocate more buffer space than is actually needed. For this reason, InterSystems IRIS supplies a system-wide default ODBC VARCHAR maximum length of 4096; this system-wide default is configurable using the Management Portal: from **System Administration**, select **Configuration**, then **SQL and Object Settings**, then **SQL**. View or set the **Default length for VARCHAR** option. To determine the current setting, call **$SYSTEM.SQL.CurrentSettings()**. The InterSystems ODBC driver takes the data from the TCP buffer and converts it into the applications buffer, so MAXLEN size does not affect our ODBC client.

Maximum and Minimum Values

In data type classes, the MINVAL and MAXVAL parameters specify the minimum and maximum values of numeric data types. Data type classes often define these values using function parameters instead of literal parameters. DDL types do not have equivalent parameters.

In this field definition, the data type is an integer from 0 to 100.

```
Capacity %Integer(MINVAL=0,MAXVAL=100)
```

Precision and Scale

In data type classes, the PRECISION and SCALE parameters are integer values specifying the precision (maximum number of digits) and scale (maximum number of decimal digits) of numeric data types. Data type classes often define these values as function parameters instead of literal parameters. DDL data types such as NUMERIC often specify precision and scale together as unnamed parameters.

This field definition defines a number that has a precision of 6 and a scale of 2.

DDL

```
UnitPrice NUMERIC(6,2) // Range: -9999.99 to 9999.99
```

The precision and scale parameters define numeric data types as follows:

- Precision — The maximum and minimum permitted value, specified as an integer from 0 to $19 + s$, where s is the scale. Precision is commonly the total number of digits in the number, but its exact value is determined by the %Library class data type mapping. The maximum integer value is 9223372036854775807. A precision larger than $19 + s$ defaults to $19 + s$.

- Scale — The maximum number of decimal (fractional) digits permitted, specified as an integer. If s is larger than or equal to the precision, p, only a fractional value is permitted, and the actual p value is ignored. The largest permitted scale is 18, which corresponds to .999999999999999999. A scale larger than 18 defaults to 18.

For information about numeric formatting, refer to the $FNUMBER function.

Function Parameters

Function parameters are used when a parameter in the DDL data type parameter must be transformed before it can be put into the InterSystems IRIS data type. An example of this is the transformation of a DDL data type's numeric precision and scale parameters into an InterSystems IRIS data type's *MAXVAL*, *MINVAL* parameters.

For example, consider the mapping between the DECIMAL DDL data type and the %Numeric class as it appears in the **System-defined DDL Mappings** table in the **Management Portal**.

DDL Using SQL Data Type

```
DECIMAL(%1,%2)
```

DDL Using InterSystems IRIS Data Type

```
%Numeric(MAXVAL=<|'$$maxval^%apiSQL(%1,%2)'|>,MINVAL=<|'$$minval^%apiSQL(%1,%2)'|>,SCALE=%2)
```

The %1 and %2 parameters specify the precision and scale of numbers in that data type, respectively. For example, a field of type DECIMAL(4,2) stores numbers using a precision of 4 and a scale of 2. InterSystems SQL uses these parameters to derive the minimum value (-99.99) and maximum value (99.99) accepted by the field.

For the %Numeric class, InterSystems IRIS set the SCALE parameter (SCALE=%2) but it does not have a PRECISION parameter to set. Instead, InterSystems IRIS sets the MAXVAL and MINVAL parameters using these transformation functions:

- **maxval^%apiSQL(*precision,scale*)** returns the maximum valid numeric value, MAXVAL, given the precision and scale.

- **minval^%apiSQL(*precision,scale*)** returns the minimum valid numeric value, MINVAL, given the precision and scale.

The syntax for these transformation functions is as follows:

```
dataTypeClass(param=<|'func'|>, param2=<|func2|>, ...)
```

- *dataTypeClass* — Name of the data type class being mapped to. Example: %Numeric

- *param* — Name of the data type class parameter being set. Example: MAXVAL

- *func* — The function call used to set the parameter. Example: maxval^%apiSQL(%1,%2)

The <|'func'|> expression signals the DDL processor to replace the parameters within func using the supplied values and then call the function with those values supplied. The <|'func'|> expression is then replaced with the value returned from the function call.

Additional Parameters

A data type class may define additional data definition parameters that cannot be defined using a DDL data type. These include data validation operations such as an enumerated list of permitted data values, pattern matching of permitted data values, and automatic truncation of data values that exceed the MAXLEN maximum length.

Create New DDL Data Types

You can modify the set of data types either by overriding the data type mapping for a system data type parameter value, or by defining a new user data type. You can modify system data types to override the InterSystems default mappings. You can create user-defined data types to provide additional data type mappings that InterSystems does not supply.

To create a new DDL data type and its mapping:

1. From the Management Portal, select **System Administration**.

2. Under **Configuration** and then **SQL and Object Settings**, click **User-defined DDL Mappings**.

3. Click **Create New User-defined DDL Mapping** to open a form for entering your data type.

4. In the **Name** field of the form, enter a DDL data type specification. For example: VARCHAR(100).

5. In the **Datatype** field, enter the name of an existing InterSystems IRIS data type class or one that you created. For example: MyString100(MAXLEN=100).

6. Click **Save**.

The **User-defined DDL Mappings** table displays the new entry. From this table, you can modify or delete the entry.

You can create a user-defined data type as a data type class. For example, you might wish to create a string data type that takes up to 10 characters and then truncates the rest of the input data. You would create this data type Sample.TruncStr, as follows:

Class Definition

```
Class Sample.TruncStr Extends %Library.String
  {
  Parameter MAXLEN=10;
  Parameter TRUNCATE=1;
  }
```

To use this data type in a table definition, specify the data type class name:

```
CREATE TABLE Sample.ShortNames (Name Sample.TruncStr)
```

When creating data type classes, keep these points in mind:

- To set parameters such as MINVAL and MAXVAL in your functions, you can use function parameters.

- If you need to map a DDL data type to an InterSystems IRIS property with a collection type of Stream, specify %Stream.GlobalCharacter for Character Stream data and %Stream.GlobalBinary for Binary Stream data.

- If DDL encounters a data type not in the DDL data type column of the **SystemDataTypes** table, it next examines the **UserDataTypes** table. If no mapping appears for the data type in either table, no conversion of the data type occurs, and the data type passes directly to the class definition as specified in DDL.

For example, the following field definitions could appear in a DDL statement:

SQL

```
    CREATE TABLE TestTable (
        Field1 %String,
        Field2 %String(MAXLEN=45)
        )
```

Given the above definitions, if DDL finds no mappings for `%String` or `%String(MAXLEN=%1)` or `%String(MAXLEN=45)` in **SystemDataTypes** or **UserDataTypes**, then the `%String` and `%String(MAXLEN=45)` types are passed directly to the appropriate class definition.

Work With Specific Data Types

Date, Time, and Timestamp Data Types

Using standard InterSystems SQL date, time functions, you can define date, time, and timestamp data types. You can also convert between dates and timestamps. For example, you can use **CURRENT_DATE** or **CURRENT_TIMESTAMP** as input to a field defined with that data type, or use **DATEADD**, **DATEDIFF**, **DATENAME**, or **DATEPART** to manipulate date values stored with this data type.

This table shows how date, time, and timestamp data type classes map to SQL types. The data type classes use this type when performing calculations in SQL. When creating a custom data type class, you can use these mappings to determine which SQL type to specify in the SqlCategory keyword of your class definition. For example:

Class Definition

```
Class MyApp.MyDateDT [ ClassType = DataType, SQLCategory = DATE ]
{
    // class members
}
```

Data Type Class	Corresponding SQL Type	Notes

Data Type Class	Corresponding SQL Type	Notes
• %Library.Date classes • Any property or column that has a logical value of +$HOROLOG (the date portion of $HOROLOG)	DATE	By default, the DATE and the corresponding %Library.Date data types accept only positive integers, with 0 representing 1840-12-31. To support dates earlier than 1840-12-31 you must define a date field in the table with data type %Library.Date(MINVAL=-nnn), where the MINVAL is a negative number of days counting backwards from 1840-12-31 to a maximum of -672045 (0001-01-01). %Library.Date can store a date value as an unsigned or negative integer in the range -672045 to 2980013. Date values can be input as follows: • Logical mode accepts +HOROLOG integer values, such as 65619 (August 28, 2020). • Display mode uses the **DisplayToLogical()** conversion method. It accepts a date in the display format for the current locale, for example '8/28/2020'. It also accepts a logical date value (a +HOROLOG integer value). • ODBC mode uses the **OdbcToLogical()** conversion method. It accepts a date in ODBC standard format, for example '2020–08–28'. It also accepts a logical date value (a +HOROLOG integer value).

Data Type Class	Corresponding SQL Type	Notes
• %Library.Time classes • Any class that has a logical value of $PIECE($HOROLOG,",",2), that is, the time portion of $HOROLOG	TIME	

Data Type Class	Corresponding SQL Type	Notes

Data Type Class	Corresponding SQL Type	Notes
		%Library.Time stores a time value as an unsigned integer in the range 0 through 86399 (a count of seconds since midnight). Time values can be input as follows:
		• Logical mode accepts $PIECE($HOROLOG,",",2) integer values, such as 84444 (23:27:24).
		• Display mode uses the **DisplayToLogical()** conversion method. It accepts a time in the display format for the current locale, for example '23:27:24'.
		• ODBC mode uses the **OdbcToLogical()** conversion method. It accepts a time in ODBC standard format, for example '23:27:24'. It also accepts a logical time value (an integer in the range 0 through 86399).
		TIME supports fractional seconds, so this data type can also be used for HH:MI:SS.FF to a user-specified number of fractional digits of precision (F), up to a maximum of 9. To support fractional seconds set the PRECISION parameter. For example, TIME(0) (`%Time(PRECISION=0)`) rounds to the nearest second; TIME(2) (`%Time(PRECISION=2)`) rounds (or zero-fills) to two fractional digits of precision.
		If the supplied data also specifies a precision (for example, **CURRENT_TIME(3)**), the fractional digits stored are as follows:
		• If TIME specifies no precision and the data specifies a precision, use the precision of the data.
		• If TIME specifies no precision and the data specifies no precision, use the system-wide configured time precision.
		• If TIME specifies a precision and the data specifies no precision, use the system-wide configured time precision as the data precision.
		• If TIME specifies a precision and the

Data Type Class	Corresponding SQL Type	Notes
		data precision is less than the TIME precision, use the data precision.
		• If TIME specifies a precision and the data precision is greater than the TIME precision, use the TIME precision.
		SQL metadata reports fractional digits of time precision as "scale"; it uses the word "precision" for the overall length of the data. A field using the TIME data type reports precision and scale metadata as follows: TIME(0) (`%Time(PRECISION=0)`) has a metadata precision of 8 (nn:nn:nn) and a scale of 0. TIME(2) (`%Time(PRECISION=2)`) has a metadata precision of 11 (nn:nn:nn.ff) and a scale of 2. TIME (`%Time` or `%Time(PRECISION="")`) take their fractional seconds of precision from the supplied data, and therefore have a metadata precision of 18 and an undefined scale. For details on returning data type, precision and scale metadata, refer to Select-item Metadata.
• %Library.TimeStamp classes • Any class that has a logical value of YYYY-MM-DD HH:MI:SS.FF	TIMESTAMP	%Library.TimeStamp derives its maximum precision from the system platform's precision, up to a maximum of 9 fractional second digits, while %Library.PosixTime has a maximum precision of 6 digits. Therefore, %Library.TimeStamp may be more precise than %Library.PosixTime on some platforms. %Library.TimeStamp normalization automatically truncates input values with more than 9 digits of precision to 9 fractional second digits.
		Note: %Library.DateTime is a subclass of %Library.TimeStamp. It defines a type parameter named DATE-FORMAT and it overrides the **DisplayToLogical()** and **OdbcToLogical()** methods to handle imprecise datetime input that TSQL applications are accustomed to.

Data Type Class	Corresponding SQL Type	Notes
• %MV.Date classes • Any class that has a logical date value of $HOROLOG-46385, which is the expression used to convert an ObjectScript date to a Multi-Value date.	MVDATE	This data type is supported only for Multi-Value compatbility.
• Any data type that does not fit into any of the preceding logical values	DATE	When defining this class, define a **LogicalToDate()** method to convert logical date values to %Library.Date logical values, and a **DateToLogical()** method that performs the reverse operation.

Data Type Class	Corresponding SQL Type	Notes
• %Library.PosixTime classes • Any user-defined data type class that has an encoded signed 64-bit integer logical value	POSIXTIME	

Data Type Class	Corresponding SQL Type	Notes
		%PosixTime is an encoded timestamp calculated from the number of seconds (and fractional seconds) since 1970–01–01 00:00:00. Timestamps after that date are represented by a positive %PosixTime value, timestamps before that date are represented by a negative %PosixTime value. %PosixTime supports a maximum of 6 digits of precision for fractional seconds. The earliest date supported by %PosixTime is 0001-01-01 00:00:00, which has a logical value of -6979664624441081856. The last date supported is 9999-12-31 23:59:59.999999, which has a logical value of 1406323805406846975.
		Because a %PosixTime value is always represented by a encoded 64-bit integer, it can always be unambiguously differentiated from a %Date or %TimeStamp value. For example, the %PosixTime value for 1970–01–01 00:00:00 is 1152921504606846976, the %PosixTime value for 2017–01–01 00:00:00 is 1154404733406846976, and the %PosixTime value for 1969–12–01 00:00:00 is -6917531706041081856.
		%PosixTime is preferable to %TimeStamp, because it takes up less disk space and memory than the %TimeStamp data type and provides better performance than %TimeStamp.
		You can integrate %PosixTime and %TimeStamp values by using the ODBC display mode:
		• Logical mode values for %PosixTime and %TimeStamp data types are completely different: %PosixTime is a signed integer, %TimeStamp is a string containing an ODBC-format timestamp.
		• Display mode: %PosixTime display uses the current locale time and date format parameters (for example, 02/22/2018 08:14:11); %TimeStamp displays as an ODBC-format timestamp.

Data Type Class	Corresponding SQL Type	Notes
		• ODBC mode: both %PosixTime and %TimeStamp display as an ODBC-format timestamp. The number of fractional digits of precision may differ.
		You can convert %TimeStamp values to %PosixTime using the TO_POSIXTIME function or the **TOPOSIXTIME()** method. You can use the **IsValid()** method to determine if a numeric value is a valid %PosixTime value.
Any time data type that does not fit into any of the preceding logical values	TIME	When creating this class, define a **LogicalToTime()** method to convert logical time values to %Library.Time logical values, and a **TimeToLogical()** method that performs the reverse operation.
Any timestamp data type that does not fit into any of the preceding logical values	TIMESTAMP	When defining this class, define a **LogicalToTimeStamp()** method to convert logical timestamp values to %Library.TimeStamp logical values, and a **TimeStampToLogical()** method that performs the reverse operation.

You can compare POSIXTIME to DATE or TIMESTAMP values using =, <>, >, or < operators. Refer to Overview of Predicates for further details.

When comparing FMTIMESTAMP category values with DATE category values, InterSystems IRIS does not strip the time from the FMTIMESTAMP value before comparing it to the DATE. This is identical behavior to comparing TIMESTAMP with DATE values, and comparing TIMESTAMP with MVDATE values. It is also compatible with how other SQL vendors compare timestamps and dates. This means a comparison of a FMTIMESTAMP 320110202.12 and DATE 62124 are equal when compared using the SQL equality (=) operator. Applications must convert the FMTIMESTAMP value to a DATE or FMDATE value to compare only the date portions of the values.

Dates Prior to December 31, 1840

A date is commonly represented by the DATE data type or the TIMESTAMP data type.

The DATE data type stores a date in $HOROLOG format, as a positive integer count of days from the arbitrary starting date of December 31, 1840. By default, dates can only be represented by a positive integer (MINVAL=0), which corresponds to the date December 31, 1840. However, you can change the %Library.Date MINVAL type parameter to enable storage of dates prior to December 31, 1840. By setting MINVAL to a negative number, you can store dates prior to December 31, 1840 as negative integers. The earliest allowed MINVAL value is -672045. This corresponds to January 1 of Year 1 (CE). DATE data type cannot represent BCE (also known as BC) dates.

The TIMESTAMP data type defaults to 1840–12–31 00:00:00 as the earliest allowed timestamp. However, you can change the MINVAL parameter to define a field or property that can store dates prior to December 31, 1840. For example, MyTS %Library.TimeStamp(MINVAL='1492-01-01 00:00:00'). The earliest allowed MINVAL value is 0001–01–01 00:00:00. This corresponds to January 1 of Year 1 (CE). The %TimeStamp data type cannot represent BCE (also known as BC) dates.

Note: Be aware that these date counts do not take into account changes in date caused by the Gregorian calendar reform (enacted October 15, 1582, but not adopted in Britain and its colonies until 1752).

You can redefine the minimum date for your locale as follows:

ObjectScript

```
SET oldMinDate = ##class(%SYS.NLS.Format).GetFormatItem("DATEMINIMUM")
IF oldMinDate=0 {
   DO ##class(%SYS.NLS.Format).SetFormatItem("DATEMINIMUM",-672045)
   SET newMinDate = ##class(%SYS.NLS.Format).GetFormatItem("DATEMINIMUM")
   WRITE "Changed earliest date to ",newMinDate
   }
ELSE { WRITE "Earliest date was already reset to ",oldMinDate}
```

The above example sets the MINVAL for your locale to the earliest permitted date (1/1/01). For more details on configuring dates based on your locale, see Configuring National Language Support (NLS).

Note: InterSystems IRIS does not support using Julian dates with negative logical DATE values (%Library.Date values with MINVAL<0). Thus, these MINVAL<0 values are not compatible with the Julian date format returned by the TO_CHAR function.

Strings

InterSystems IRIS permits a fixed amount of memory to handle strings, so that there is a string length limit. Commonly, extremely long strings should be assigned one of the %Stream.GlobalCharacter data types.

No string length limit is enforced over a database driver connection. If the InterSystems IRIS instance and the ODBC driver facilities support different protocols, the lower of the two protocols is used. The protocol that was actually used is recorded in the InterSystems ODBC log.

Note that, by default, InterSystems IRIS establishes a system-wide ODBC VARCHAR maximum length of 4096; this ODBC maximum length is configurable.

List Structures

InterSystems IRIS supports the list structure data type %List (data type class %Library.List). This is a compressed binary format, which does not map to a corresponding native data type for InterSystems SQL. In its internal representation it corresponds to data type VARBINARY with a default MAXLEN of 32749. InterSystems IRIS supports the list structure data type %ListOfBinary (data type class %Library.ListOfBinary) corresponds to data type VARBINARY with a default MAXLEN of 4096.

For this reason, Dynamic SQL cannot use %List data in a **WHERE** clause comparison. You also cannot use **INSERT** or **UPDATE** to set a property value of type %List.

Dynamic SQL returns the data type of list structured data as VARCHAR. To determine if a field in a query is of data type %List or %ListOfBinary you can use the select-item columns metadata `isList` boolean flag. The CType (client data type) integer code for these data types is 6.

If you use an ODBC or JDBC client, %List data is projected to VARCHAR string data, using LogicalToOdbc conversion. A list is projected as a string with its elements delimited by commas. Data of this type can be used in a **WHERE** clause, and in **INSERT** and **UPDATE** statements. Note that, by default, InterSystems IRIS establishes a system-wide ODBC VARCHAR maximum length of 4096; this ODBC maximum length is configurable.

Also see %Library.List for information on that class. For further details on using lists in a **WHERE** clause, see the %INLIST predicate and the FOR SOME %ELEMENT predicate. For further details on handling list data as a string, see the %EXTERNAL function.

InterSystems SQL supports eight list functions: $LIST, $LISTBUILD, $LISTDATA, $LISTFIND, $LISTFROMSTRING, $LISTGET, $LISTLENGTH, and $LISTTOSTRING. ObjectScript supports three additional list functions: $LISTVALID

to determine if an expression is a list, $LISTSAME to compare two lists, and $LISTNEXT to sequentially retrieve elements from a list.

BIT Data Type

The BIT (%Library.Boolean) data type accepts 0, 1, and NULL as valid values.

- In Logical and ODBC modes the only accepted values are 0, 1, and NULL.

- In Display mode the DisplayToLogical method first translates a non-null input value to 0 or 1, as follows:

  - Non-zero numbers or numeric strings = 1. For example, 3, '0.1', '-1', '7dwarves'.

  - Non-numeric strings = 0. For example, 'true' or 'false'.

  - Empty string = 0. For example, ''.

Stream Data Types

The Stream data types correspond to the InterSystems IRIS class property data types %Stream.GlobalCharacter (for CLOBs) and %Stream.GlobalBinary (for BLOBs). These data type classes can define a stream field with a specified LOCATION parameter, or omit this parameter and default to a system-defined storage location.

A field with a Stream data type cannot be used as an argument to most SQL scalar, aggregate, or unary functions. Attempting to do so generates an SQLCODE -37 error code. The few functions that are exceptions are listed in Storing and Using Stream Data (BLOBs and CLOBs).

A field with a Stream data type cannot be used as an argument to most SQL predicate conditions. Attempting to do so generates an SQLCODE -313 error code. The predicates that accept a stream field are listed in Storing and Using Stream Data (BLOBs and CLOBs).

A sharded table cannot contain stream data type fields.

The use of Stream data types in indexes, and when performing inserts and updates are also restricted. For further details on Stream restrictions, refer to Storing and Using Stream Data (BLOBs and CLOBs).

SERIAL Data Type

A field with a SERIAL (%Library.Counter) data type can take a user-specified positive integer value, or InterSystems IRIS can assign it a sequential positive integer value. %Library.Counter extends %Library.BigInt.

An **INSERT** operation specifies one of the following values for a SERIAL field:

- No value, 0 (zero), or a nonnumeric value: InterSystems IRIS ignores the specified value, and instead increments this field's current serial counter value by 1, and inserts the resulting integer into the field.

- A positive integer value: InterSystems IRIS inserts the user-specified value into the field, and changes the serial counter value for this field to this integer value.

Thus a SERIAL field contains a series incremental integer values. These values are not necessarily continuous or unique. For example, the following is a valid series of values for a SERIAL field: 1, 2, 3, 17, 18, 25, 25, 26, 27. Sequential integers are either InterSystems IRIS-generated or user-supplied; nonsequential integers are user-supplied. If you wish SERIAL field values to be unique, you must apply a UNIQUE constraint on the field.

An **UPDATE** operation has no effect on automatically-assigned SERIAL counter field values. However, an update performed using INSERT OR UPDATE causes a skip in integer sequence for subsequent insert operations for a SERIAL field.

An **UPDATE** operation can only change a serial field value if the field currently has no value (NULL), or its value is 0. Otherwise, an SQLCODE -105 error is generated.

InterSystems IRIS imposes no restriction on the number of SERIAL fields in a table.

ROWVERSION Data Type

The ROWVERSION data type defines a read-only field that contains a unique system-assigned positive integer, beginning with 1. InterSystems IRIS assigns sequential integers as part of each insert, update, or %Save operation. These values are not user-modifiable.

InterSystems IRIS maintains a single row version counter namespace-wide. All tables in a namespace that contain a ROWVERSION field share the same row version counter. Thus, the ROWVERSION field provides row-level version control, allowing you to determine the order in which changes were made to rows in one or more tables in a namespace.

You can only specify one field of ROWVERSION data type per table.

The ROWVERSION field should not be included in a unique key or primary key. The ROWVERSION field cannot be part of an IDKey index.

For details on using ROWVERSION, refer to RowVersion Field.

ROWVERSION and SERIAL Counters

Both ROWVERSION and SERIAL (%Library.Counter) data type fields receive a sequential integer from an internal counter as part of an **INSERT** operation. But these two counters are significantly different and are used for different purposes:

- The ROWVERSION counter is at the namespace level. The SERIAL counter is at the table level. These two counters are completely independent of each other and independent of the RowID counter.

- The ROWVERSION counter is incremented by insert, update, or %Save operations. The SERIAL counter is only incremented by insert operations. An update performed using **INSERT OR UPDATE** can cause a gap in the SERIAL counter sequence.

- A ROWVERSION field value cannot be user-specified; the value is always supplied from the ROWVERSION counter. A SERIAL field value is supplied from the table's internal counter during an insert if you do not specify a value for this field. If an insert supplies a SERIAL integer value, that value is inserted rather than the current counter value:

  - If an insert supplies a SERIAL field value greater than the current internal counter value, InterSystems IRIS inserts that value into the field and resets the internal counter to that value.

  - If an insert supplies a SERIAL field value less than the current counter value, InterSystems IRIS does not reset the internal counter.

  - An insert can supply a SERIAL field value as a negative integer or a fractional number. InterSystems IRIS truncates a fractional number to its integer component. If the supplied SERIAL field value is 0 or NULL, InterSystems IRIS ignores the user-supplied value and inserts the current internal counter value.

  You cannot update an existing SERIAL field value.

- A ROWVERSION field value is always unique. Because you can insert a user-specified SERIAL field value, you must specify a UNIQUE field constraint to guarantee unique SERIAL field values.

- The ROWVERSION counter cannot be reset. A **TRUNCATE TABLE** resets the SERIAL counter; performing a **DELETE** on all rows does not reset the SERIAL counter.

- Only one ROWVERSION field is allowed per table. You can specify multiple SERIAL fields in a table.

DDL Data Types Exposed by InterSystems ODBC / JDBC

InterSystems ODBC exposes a subset of the DDL data types, and maps other data types to this subset of data types. These mappings are not reversible. For example, the statement `CREATE TABLE mytable (f1 BINARY)` creates an InterSystems IRIS class that is projected to ODBC as `mytable (f1 VARBINARY)`. An InterSystems IRIS list data type is projected to ODBC as a VARCHAR string.

ODBC exposes the following data types: BIGINT, BIT, DATE, DOUBLE, GUID, INTEGER, LONGVARBINARY, LONGVARCHAR, NUMERIC, OREF, POSIXTIME, SMALLINT, TIME, TIMESTAMP, TINYINT, VARBINARY, VARCHAR. Note that, by default, InterSystems IRIS establishes a system-wide ODBC VARCHAR maximum length of 4096; this ODBC maximum length is configurable.

When one of these ODBC/JDBC data type values is mapped to InterSystems SQL, the following operations occur: DOUBLE data is cast using $DOUBLE. NUMERIC data is cast using $DECIMAL.

The GUID data type corresponds to InterSystems SQL UNIQUEIDENTIFIER data type. Failing to specify a valid value to a GUID / UNIQUEIDENTIFIER field generates a #7212 General Error. To generate a GUID value, use the **%SYSTEM.Util.CreateGUID()** method.

Convert Between Data Types

To convert data from one data type to another, use the CAST or CONVERT function.

CAST supports conversion to several character string and numeric data types, as well as to DATE, TIME, and the TIMESTAMP and POSIXTIME timestamp data types.

CONVERT has two syntactical forms. Both forms support conversion to and from DATE, TIME, and the TIMESTAMP and POSIXTIME timestamp data types, as well as conversion between other data types.

When you **CAST** or **CONVERT** a value to VARCHAR, the default size mapping is 30 characters, even though VARCHAR with no specified size maps to a MAXLEN of 1. This default size of 30 characters is provided for compatibility with non-InterSystems IRIS software requirements.

Data Type Precedence

When an operation can return several different values, and these values may have different data types, InterSystems IRIS assigns the return value whichever data type has the highest precedence. For example, a NUMERIC data type can contain all possible INTEGER data type values, but an INTEGER data type cannot contain all possible NUMERIC data type values. Thus NUMERIC has the higher precedence (is more inclusive).

For example, if a CASE statement has a possible result value of data type INTEGER, and a possible result value of data type NUMERIC, the actual result is always of type NUMERIC, regardless of which of these two cases are taken.

The precedence for data types is as follows, from highest (most inclusive) to lowest:

```
LONGVARBINARY
LONGVARCHAR
VARBINARY
VARCHAR
GUID
TIMESTAMP
DOUBLE
NUMERIC
BIGINT
INTEGER
DATE
TIME
SMALLINT
TINYINT
BIT
```

Data Type Normalization and Validation

The %Library.DataType superclass has subclasses for the specific data types. These data type classes provide a **Normalize()** method to normalize an input value to the data type format and an **IsValid()** method to determine if an input value is valid for that data type, as well as various mode conversion methods such as **LogicalToDisplay()** and **DisplayToLogical()**.

The following examples show the **Normalize()** method for the %TimeStamp data type:

ObjectScript

```
SET indate=64701
SET tsdate=##class(%Library.TimeStamp).Normalize(indate)
WRITE "%TimeStamp date: ",tsdate
```

ObjectScript

```
SET indate="2018-2-22"
SET tsdate=##class(%Library.TimeStamp).Normalize(indate)
WRITE "%TimeStamp date: ",tsdate
```

The following examples show the **IsValid()** method for the %TimeStamp data type:

ObjectScript

```
SET datestr="July 4, 2018"
SET stat=##class(%Library.TimeStamp).IsValid(datestr)
  IF stat=1 {WRITE datestr," is a valid %TimeStamp",! }
  ELSE {WRITE datestr," is not a valid %TimeStamp",!}
```

ObjectScript

```
SET leapdate="2016-02-29 00:00:00"
SET noleap="2018-02-29 00:00:00"
SET stat=##class(%Library.TimeStamp).IsValid(leapdate)
  IF stat=1 {WRITE leapdate," is a valid %TimeStamp",! }
  ELSE {WRITE leapdate," is not a valid %TimeStamp",!}
SET stat=##class(%Library.TimeStamp).IsValid(noleap)
  IF stat=1 {WRITE noleap," is a valid %TimeStamp",! }
  ELSE {WRITE noleap," is not a valid %TimeStamp",!}
```

Returning Data Types Using Query Metadata

You can use Dynamic SQL to return metadata about a query, including the data type of a specified column in the query.

The following Dynamic SQL examples return the column name and the integer code for the ODBC data type for each of the columns in Sample.Person and Sample.Employee:

ObjectScript

```
SET myquery="SELECT * FROM Sample.Person"
SET tStatement=##class(%SQL.Statement).%New()
SET tStatus=tStatement.%Prepare(myquery)
  SET x=tStatement.%Metadata.columnCount
WHILE x>0 {
  SET column=tStatement.%Metadata.columns.GetAt(x)
WRITE !,x," ",column.colName," ",column.ODBCType
SET x=x-1 }
WRITE !,"end of columns"
```

ObjectScript

```
SET myquery="SELECT * FROM Sample.Employee"
SET tStatement=##class(%SQL.Statement).%New()
SET tStatus=tStatement.%Prepare(myquery)
  SET x=tStatement.%Metadata.columnCount
WHILE x>0 {
  SET column=tStatement.%Metadata.columns.GetAt(x)
WRITE !,x," ",column.colName," ",column.ODBCType
SET x=x-1 }
WRITE !,"end of columns"
```

List structured data, such as the FavoriteColors column in Sample.Person, returns a data type of 12 (VARCHAR) because ODBC represents an ObjectScript %List data type value as a string of comma-separated values.

Steams data, such as the Notes and Picture columns in Sample.Employee, return the data types -1 (LONGVARCHAR) or -4 (LONGVARBINARY).

A ROWVERSION field returns data type -5 because %Library.RowVersion is a subclass of %Library.BigInt.

For further details, refer to Dynamic SQL and see %SQL.Statement.

Integer Codes for Data Types

In query metadata and other contexts, the defined data type for a column may be returned as an integer code. The CType (client data type) integer codes are listed in the %SQL.StatementColumn *clientType* property. For further details, refer to Select-item Metadata.

SQLType data type codes are used by ODBC and JDBC. ODBC data type codes are returned by **%SQL.Statement.%Metadata.columns.GetAt()** method, as shown in the example above. SQL Shell metadata also returns ODBC data type codes. The JDBC codes are the same as the ODBC codes, except in the representation of time and date data types. These ODBC and JDBC values are listed below:

ODBC	JDBC	Data Type
-11	-11	GUID
-7	-7	BIT
-6	-6	TINYINT
-5	-5	BIGINT
-4	-4	LONGVARBINARY
-3	-3	VARBINARY
-2	-2	BINARY
-1	-1	LONGVARCHAR
0	0	Unknown type
1	1	CHAR
2	2	NUMERIC
3	3	DECIMAL
4	4	INTEGER
5	5	SMALLINT
6	6	FLOAT
7	7	REAL
8	8	DOUBLE
9	91	DATE
10	92	TIME
11	93	TIMESTAMP
12	12	VARCHAR

For further details, refer to Dynamic SQL.

InterSystems IRIS also supports Unicode SQL types for ODBC applications working with multibyte character sets, such as in Chinese, Hebrew, Japanese, or Korean locales.

ODBC	Data Type
-10	WLONGVARCHAR
-9	WVARCHAR

To activate this functionality, refer to Using an InterSystems Database as an ODBC Data Source on Windows.

See Also

- CAST, CONVERT

- TO_CHAR, TO_DATE, TO_NUMBER

Date and Time Constructs (SQL)

Validates and converts an ODBC date, time, or timestamp.

Synopsis

```
{d 'yyyy-mm-dd'}
{d nnnnnn}

{t 'hh:mm:ss[.fff]'}
{t nnnnn.nnn}

{ts 'yyyy-mm-dd [hh:mm:ss.fff]'}
{ts 'mm/dd/yyyy [hh:mm:ss.fff]'}
{ts nnnnnn}
```

Description

These constructs take either an integer or a string in ODBC date, time, or timestamp format and convert it to the corresponding InterSystems IRIS date, time, or timestamp format. They perform data typing and value and range checking.

{d 'string'}

The {d 'string'} date construct validates a date in ODBC format. If the date is valid, it stores it (logical mode) in InterSystems IRIS $HOROLOG date format as an integer count value from 1840-12-31. InterSystems IRIS does not append a default time value. To support dates earlier than 1840-12-31 you must define the date field in the table with data type %Library.Date(MINVAL=-nnn), where the MINVAL is a negative number of days counting backwards from 1840-12-31 (day 0) to a maximum of -672045 (0001-01-01).

If you supply:

- An integer less than -672045 (0001-01-01) or greater than 2980013 (9999-12-31) generates an SQLCODE -400 <VALUE OUT OF RANGE> error.

- An invalid date (such as a date not in ODBC format or the date 02-29 in a non-leap year): InterSystems IRIS generates an SQLCODE -146 error: "yyyy-mm-dd' is an invalid ODBC/JDBC Date value".

- An ODBC timestamp value: InterSystems IRIS validates both the date and time portions of the timestamp. If both are valid, it converts the date portion only. If either date or time are invalid, the system generates an SQLCODE -146 error.

{t 'string'}

The {t 'string'} time construct validates a time in ODBC format. If the time is valid, it stores it (logical mode) in InterSystems IRIS $HOROLOG time format as an integer count of seconds from midnight, with the specified fractional seconds. Inter-Systems IRIS Display mode and ODBC mode do not display the fractional seconds; the fractional seconds are truncated from these display formats.

If you supply:

- An integer less than 0 (00:00:00) or greater than 86399.99 (23:59:59.99) generates an SQLCODE -400 <ILLEGAL VALUE> error.

- An invalid time (such as a time not in ODBC format or a time with hour >23): InterSystems IRIS generates an SQLCODE -147 error: "hh:mi:ss.fff' is an invalid ODBC/JDBC Time value".

- An ODBC timestamp value: InterSystems IRIS generates an SQLCODE -147 error.

{ts 'string'}

The {ts 'string'} timestamp construct validates a date/time and returns it in ODBC timestamp format; specified fractional seconds are always preserved and displayed. The {ts 'string'} timestamp construct also validates a date and returns it in ODBC timestamp format with a suppled time value of 00:00:00.

If you supply:

- A positive or negative integer date (-672045 through 2980013): InterSystems IRIS appends a time value of 00:00:00, then stores the resulting timestamp in ODBC format. For example, `64701` returns 2018-02-22 00:00:00. This is a valid $HOROLOG date integer. $HOROLOG 0 is 1840-12-31.

- A valid timestamp in ODBC format: InterSystems IRIS stores the supplied value unchanged This is because InterSystems IRIS timestamp format is the same as ODBC timestamp format.

- A valid timestamp using the locale default date and time formats (for example, 2/29/2016 12:23:46.77): InterSystems IRIS stores and displays the supplied value in ODBC format.

- An invalid timestamp (such as a timestamp with the date portion specifying 02-29 in a non-leap year, or with the time portion specifying hour >23): InterSystems IRIS returns the string "`error`" as the value.

- A valid date (in ODBC or locale format) with no time value: InterSystems IRIS appends a time value of 00:00:00, then stores the resulting timestamp in ODBC format. It supplies leading zeros where necessary. For example, `2/29/2016` returns 2016-02-29 00:00:00.

- A correctly formatted, but invalid, date (in ODBC or locale format) with no time value: InterSystems IRIS appends a time value of 00:00:00. It then stores the date portion as supplied. For example, `02/29/2019` returns 02/29/2019 00:00:00.

- An incorrectly formatted and invalid, date (in ODBC, locale, or $HOROLOG format) with no time value: InterSystems IRIS returns the string "`error`". For example, `2/29/2019` (no leading zero and invalid date value) returns "`error`". `00234` ($HOROLOG with leading zeros) returns "`error`"

See $HOROLOG for further information.

Examples

The following Dynamic SQL example validates dates supplied in ODBC format (with or without leading zeros) and stores them as the equivalent $HOROLOG value 64701. This example displays %SelectMode 0 (logical) values:

ObjectScript

```
SET myquery = 2
SET myquery(1) = "SELECT {d '2018-02-22'} AS date1,"
SET myquery(2) = "{d '2018-2-22'} AS date2"
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=0
SET tStatus = tStatement.%Prepare(.myquery)
SET rset = tStatement.%Execute()
DO rset.%Display()
```

The following Dynamic SQL example validates times supplied in ODBC format (with or without leading zeros) and stores them as the equivalent $HOROLOG value 43469. This example displays %SelectMode 0 (logical) values:

ObjectScript

```
SET myquery = 3
SET myquery(1) = "SELECT {t '12:04:29'} AS time1,"
SET myquery(2) = "{t '12:4:29'} AS time2,"
SET myquery(3) = "{t '12:04:29.00000'} AS time3"
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=0
SET tStatus = tStatement.%Prepare(.myquery)
SET rset = tStatement.%Execute()
DO rset.%Display()
```

The following Dynamic SQL example validates times supplied in ODBC format with fractional seconds, and stores them as the equivalent $HOROLOG value 43469 with the fractional seconds appended. Trailing zeros are truncated. This example displays %SelectMode 0 (logical) values:

ObjectScript

```
SET myquery = 3
SET myquery(1) = "SELECT {t '12:04:29.987'} AS time1,"
SET myquery(2) = "{t '12:4:29.987'} AS time2,"
SET myquery(3) = "{t '12:04:29.987000'} AS time3"
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=0
SET tStatus = tStatement.%Prepare(.myquery)
SET rset = tStatement.%Execute()
DO rset.%Display()
```

The following Dynamic SQL example validates time and date values in several formats and stores them as the equivalent ODBC timestamp. A time value of 00:00:00 is supplied when necessary. This example displays %SelectMode 0 (logical) values:

ObjectScript

```
SET myquery = 6
SET myquery(1) = "SELECT {ts '2018-02-22 01:43:38'} AS ts1,"
SET myquery(2) = "{ts '2018-02-22'} AS ts2,"
SET myquery(3) = "{ts '02/22/2018 01:43:38.999'} AS ts3,"
SET myquery(4) = "{ts '2/22/2018 01:43:38'} AS ts4,"
SET myquery(5) = "{ts '02/22/2018'} AS ts5,"
SET myquery(6) = "{ts '64701'} AS ts6"
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=0
SET tStatus = tStatement.%Prepare(.myquery)
SET rset = tStatement.%Execute()
IF rset.%Next() {
WRITE rset.ts1,!
WRITE rset.ts2,!
WRITE rset.ts3,!
WRITE rset.ts4,!
WRITE rset.ts5,!
WRITE rset.ts6
}
```

Default user name and password (SQL)

Provides default login identity.

Description

The InterSystems IRIS® data platform provides a default user name and password for logging in to the database and getting started. The default user name is "_SYSTEM" (uppercase) and "SYS" is its password.

SQLCODE Error Codes

SQL error codes.

Description

Attempting to execute most InterSystems SQL operations issues an SQLCODE value. The SQLCODE values issued are 0, 100, and negative integer values.

- SQLCODE=0 indicates successful completion of an SQL operation. For a **SELECT** statement, this usually means the successful retrieval of data from a table. However, if the **SELECT** performs an aggregate operation, (for example: SELECT SUM(myfield)) the aggregate operation is successful and an SQLCODE=0 is issued even when there is no data in *myfield*; in this case **SUM** returns NULL and %ROWCOUNT=1.

- SQLCODE=100 indicates that the SQL operation was successful, but found no data to act upon. This can occur for a number of reasons. For a **SELECT** these include: the specified table contains no data; the table contains no data that satisfies the query criteria; or row retrieval has reached the final row of the table. For an **UPDATE** or **DELETE** these include: the specified table contains no data; or the table contains no row of data that satisfies the **WHERE** clause criteria. In these cases %ROWCOUNT=0.

- SQLCODE=-*n* indicates an error. The negative integer value specifies the kind of error that occurred. SQLCODE=-400 is a general purpose fatal error code.

For further details on SQLCODE error codes and the corresponding error messages, and a full list of SQLCODE error code values, refer to SQL Error Messages.

Field constraint

Specifies rules about a field's contents.

Description

A field constraint specifies rules governing the data values permitted for a field. A field may have the following constraints:

- NOT NULL: You must specify a value for this field in every record (empty strings acceptable).

- UNIQUE: If you specify a value for this field in a record, it must be a unique value (one empty string acceptable). You can, however, create multiple records with no value (NULL) for the field.

- DEFAULT: You must either specify a value or InterSystems IRIS provides a default for this field in every record (empty strings acceptable). The default may be NULL, an empty string, or any other value appropriate for the data type.

- UNIQUE NOT NULL: You must specify a unique value for this field in every record (one empty string acceptable). Can be used as a primary key.

- DEFAULT NOT NULL: You must either specify a value or InterSystems IRIS provides a default value for this field in every record (empty strings acceptable).

- UNIQUE DEFAULT: *Not Recommended* — You must either specify a unique value or InterSystems IRIS provides a default value for this field in every record (one empty string acceptable). The default may be NULL, an empty string, or any other value appropriate for the data type. Use only if the default is a unique generated value (for example, CURRENT_TIMESTAMP), or if the default is intended to be used only once.

- UNIQUE DEFAULT NOT NULL: *Not Recommended* — You must either specify a unique value or InterSystems IRIS provides a default value for this field in every record (one empty string acceptable). The default may be an empty string or any other value appropriate for the data type; it cannot be NULL. Use only if the default is a unique generated value (for example, CURRENT_TIMESTAMP), or if the default is intended to be used only once. Can be used as a primary key.

- IDENTITY: InterSystems IRIS provides a unique, system-generated, non-modifiable integer value for this field in every record. Other field constraint keywords are ignored. Can be used as a primary key.

Data values must be appropriate for the field's data type. An empty string is not an acceptable value for a numeric field.

These field constraints are further described in the page for the CREATE TABLE command.

Reserved words (SQL)

A list of SQL reserved words for InterSystems IRIS® data platform.

Synopsis

```
%AFTERHAVING  |  %ALLINDEX  |  %ALPHAUP  |  %ALTER  |  %BEGTRANS  |
%CHECKPRIV  |  %CLASSNAME  |  %CLASSPARAMETER  |  %DBUGFULL  |  %DELDATA  |
%DESCRIPTION  |  %EXACT  |  %EXTERNAL  |  %FILE  |  %FIRSTTABLE  |  %FLATTEN  |
%FOREACH  |  %FULL  |  %ID  |  %IDADDED  |  %IGNOREINDEX  |  %IGNOREINDICES  |
%INLIST  |  %INORDER  |  %INTERNAL  |  %INTEXT  |  %INTRANS  |  %INTRANSACTION  |
%KEY  |  %MATCHES  |  %MCODE  |  %MERGE  |  %MINUS  |  %MVR  |  %NOCHECK  |
%NODELDATA  |  %NOFLATTEN  |  %NOFPLAN  |  %NOINDEX  |  %NOLOCK  |
%NOMERGE  |  %NOPARALLEL  |  %NOREDUCE  |  %NORUNTIME  |  %NOSVSO  |  %NOTOPOPT  |
%NOTRIGGER  |  %NOUNIONOROPT  |  %NUMROWS  |  %ODBCIN  |  %ODBCOUT  |
%PARALLEL  |  %PLUS  |  %PROFILE  |  %PROFILE_ALL  |  %PUBLICROWID  |  %ROUTINE  |
%ROWCOUNT  |  %RUNTIMEIN  |  %RUNTIMEOUT  |  %STARTSWITH  |
%STARTTABLE  |  %SQLSTRING  |  %SQLUPPER  |  %STRING  |  %TABLENAME  |
%TRUNCATE  |  %UPPER  |  %VALUE  |  %VID  |
ABSOLUTE  |  ADD  |  ALL  |  ALLOCATE  |  ALTER  |  AND  |  ANY  |  ARE  |  AS  |
ASC  |  ASSERTION  |  AT  |  AUTHORIZATION  |  AVG  |  BEGIN  |  BETWEEN  |
BIT  |  BIT_LENGTH  |  BOTH  |  BY  |  CASCADE  |  CASE  |  CAST  |
CHAR  |  CHARACTER  |  CHARACTER_LENGTH  |  CHAR_LENGTH  |
CHECK  |  CLOSE  |  COALESCE  |  COLLATE  |  COMMIT  |  CONNECT  |
CONNECTION  |  CONSTRAINT  |  CONSTRAINTS  |  CONTINUE  |  CONVERT  |
CORRESPONDING  |  COUNT  |  CREATE  |  CROSS  |  CURRENT  |
CURRENT_DATE  |  CURRENT_TIME  |  CURRENT_TIMESTAMP  |
CURRENT_USER  |  CURSOR  |  DATE  |  DEALLOCATE  |  DEC  |  DECIMAL  |
DECLARE  |  DEFAULT  |  DEFERRABLE  |  DEFERRED  |  DELETE  |  DESC  |
DESCRIBE  |  DESCRIPTOR  |  DIAGNOSTICS  |  DISCONNECT  |  DISTINCT  |
DOMAIN  |  DOUBLE  |  DROP  |  ELSE  |  END  |  ENDEXEC  |  ESCAPE  |  EXCEPT  |
EXCEPTION  |  EXEC  |  EXECUTE  |  EXISTS  |  EXTERNAL  |  EXTRACT  |
FALSE  |  FETCH  |  FIRST  |  FLOAT  |  FOR  |  FOREIGN  |  FOUND  |  FROM  |  FULL  |
GET  |  GLOBAL  |  GO  |  GOTO  |  GRANT  |  GROUP  |  HAVING  |  HOUR  |
IDENTITY  |  IMMEDIATE  |  IN  |  INDICATOR  |  INITIALLY  |
INNER  |  INPUT  |  INSENSITIVE  |  INSERT  |  INT  |  INTEGER  |  INTERSECT  |
INTERVAL  |  INTO  |  IS  |  ISOLATION  |  JOIN  |  LANGUAGE  |  LAST  |
LEADING  |  LEFT  |  LEVEL  |  LIKE  |  LOCAL  |  LOWER  |  MATCH  |  MAX  |  MIN  |
MINUTE  |  MODULE  |  NAMES  |  NATIONAL  |  NATURAL  |  NCHAR  |
NEXT  |  NO  |  NOT  |  NULL  |  NULLIF  |  NUMERIC  |  OCTET_LENGTH  |  OF  |  ON  |
ONLY  |  OPEN  |  OPTION  |  OR  |  OUTER  |  OUTPUT  |  OVERLAPS  |
PAD  |  PARTIAL  |  PREPARE  |  PRESERVE  |  PRIMARY  |  PRIOR  |  PRIVILEGES  |
PROCEDURE  |  PUBLIC  |  READ  |  REAL  |  REFERENCES  |  RELATIVE  |
RESTRICT  |  REVOKE  |  RIGHT  |  ROLE  |  ROLLBACK  |  ROWS  |
SCHEMA  |  SCROLL  |  SECOND  |  SECTION  |  SELECT  |  SESSION_USER  |
SET  |  SHARD  |  SMALLINT  |  SOME  |  SPACE  |  SQLERROR  |  SQLSTATE  |
STATISTICS  |  SUBSTRING  |  SUM  |  SYSDATE  |  SYSTEM_USER  |  TABLE  |
TEMPORARY  |  THEN  |  TIME  |  TIMEZONE_HOUR  |  TIMEZONE_MINUTE  |
TO  |  TOP  |  TRAILING  |  TRANSACTION  |  TRIM  |  TRUE  |  UNION  |  UNIQUE  |
UPDATE  |  UPPER  |  USER  |  USING  |  VALUES  |  VARCHAR  |  VARYING  |  WHEN  |
WHENEVER  |  WHERE  |  WITH  |  WORK  |  WRITE
```

Description

Within SQL certain words are *reserved*. You cannot use an SQL reserved word as an SQL identifier (such as the name for a table, a column, an AS alias, or other entity), unless:

- The word is delimited with double quotes ("*word*"), *and*

- Delimited identifiers are supported. For further details, refer to Identifiers.

This list contains only those words that are reserved in this sense; it does not contain all SQL keywords. Several of the words listed above start with the "%" character, indicating that they are InterSystems SQL proprietary extension keywords. In general, it is not recommended to use words that begin with "%" as identifiers such as table and column names, because new InterSystems SQL extension keywords may be added in the future.

You can check if a word is an SQL reserved word by invoking the **IsReservedWord()** method, as shown in the following example. Specify the reserved word as a quoted string; reserved words are not case-sensitive. **$SYSTEM.SQL.IsReservedWord()** returns a boolean value.

ObjectScript

```
WRITE !,"Reserved?: ",$SYSTEM.SQL.IsReservedWord("VARCHAR")
WRITE !,"Reserved?: ",$SYSTEM.SQL.IsReservedWord("varchar")
WRITE !,"Reserved?: ",$SYSTEM.SQL.IsReservedWord("VarChar")
WRITE !,"Reserved?: ",$SYSTEM.SQL.IsReservedWord("FRED")
```

This method can also be called as a stored procedure from ODBC or JDBC: %SYSTEM.SQL_IsReservedWord("nnnn").

Special Variables

System-supplied variables.

Synopsis

```
$HOROLOG
$JOB
$NAMESPACE
$TLEVEL
$USERNAME
$ZHOROLOG
$ZJOB
$ZPI
$ZTIMESTAMP
$ZTIMEZONE
$ZVERSION
```

Description

InterSystems SQL directly supports a number of the ObjectScript special variables. These variables contain system-supplied values. They can be used wherever a literal value can be specified in InterSystems SQL.

SQL special variable names are not case-sensitive. Most can be specified using an abbreviation.

Variable Name	Abbreviation	Data Type Returned	Use
$HOROLOG	$H	%String/VARCHAR	Local date and time for the current process
$JOB	$J	%String/VARCHAR	Job ID of the current process
$NAMESPACE	none	%String/VARCHAR	Current namespace name
$TLEVEL	$TL	%Integer/INTEGER	Current transaction nesting level
$USERNAME	none	%String/VARCHAR	User name for the current process
$ZHOROLOG	$ZH	%Numeric/NUMERIC(21,6)	Number of elapsed seconds since InterSystems IRIS startup
$ZJOB	$ZJ	%Integer/INTEGER	Job status for the current process
$ZPI	none	%Numeric/NUMERIC(21,18)	The numeric constant PI
$ZTIMESTAMP	$ZTS	%String/VARCHAR	Current date and time in Coordinated Universal Time format
$ZTIMEZONE	$ZTZ	%Integer/INTEGER	Local time zone offset from GMT
$ZVERSION	$ZV	%String/VARCHAR	The current version of InterSystems IRIS

For further details, refer to the corresponding ObjectScript special variable, as described in the ObjectScript Reference.

Examples

The following example returns a result set that includes the current date and time:

SQL

```
SELECT TOP 5 Name,$H
FROM Sample.Person
```

The following example only returns a result set if the time zone is within the continental United States:

SQL

```
SELECT TOP 5 Name,Home_State
FROM Sample.Person
WHERE $ZTIMEZONE BETWEEN 300 AND 480
```

String Manipulation (SQL)

String manipulation functions and operators.

Description

InterSystems SQL provides support for several types of string manipulation:

- Strings can be manipulated by length, character position, or substring value.

- Strings can be manipulated by a designated delimiter character or delimiter string.

- Strings can tested by pattern matching and word-aware searches.

- Specially encoded strings, called lists, contain embedded substring identifiers without using a delimiter character. The various **$LIST** functions operate on these encoded character strings, which are incompatible with standard character strings. The only exceptions are the **$LISTGET** function and the one-argument and two-argument forms of **$LIST**, which take an encoded character string as input, but output a single element value as a standard character string.

InterSystems SQL supports string functions, string condition expressions, and string operators.

ObjectScript string manipulation is case-sensitive. Letters in strings can be converted to uppercase, to lowercase, or retained as mixed case. String collation can be case-sensitive, or not case-sensitive; by default, SQL string collation is SQLUPPER which is not case-sensitive. InterSystems SQL provides numerous letter case and collation functions and operators.

When a string is specified for a numeric argument, most InterSystems SQL functions perform the following string-to-number conversions: a nonnumeric string is converted to the number 0; a numeric string is converted to a canonical number; and a mixed-numeric string is truncated at the first nonnumeric character and then converted to a canonical number.

String Concatenation

The following functions concatenate substrings into a string:

- CONCAT: concatenates two substrings, returns a single string.

- STRING: concatenates two or more substrings, returns a single string.

- XMLAGG: concatenates all of the values of a column, returns a single string. For further details, see Aggregate Functions.

- LIST: concatenates all of the values of a column, including a comma delimiter, returns a single string. For further details, see Aggregate Functions.

The concatenate operator (||) can also be used to concatenate two strings.

String Length

The following functions can be used to determine the length of a string:

- CHARACTER_LENGTH and CHAR_LENGTH: return the number of characters in a string, including trailing blanks. NULL returns NULL.

- LENGTH: returns the number of characters in a string, excluding trailing blanks. NULL returns NULL.

- $LENGTH: returns the number of characters in a string, including trailing blanks. NULL is returned as 0.

Truncation and Trimming

The following functions can be used to truncate or trim a string. Truncation limits the length of the string, deleting all characters beyond the specified length. Trimming deletes leading and/or trailing blank spaces from a string.

- Truncation: CONVERT, %SQLSTRING, and %SQLUPPER.

- Trimming: TRIM, LTRIM, and RTRIM.

Substring Search

The following functions search for a substring within a string and return a string position:

- POSITION: searches by substring value, finds first match, returns position of beginning of substring.

- CHARINDEX: searches by substring value, finds first match, returns position of beginning of substring. Starting point can be specified.

- $FIND: searches by substring value, finds first match, returns position of end of substring. Starting point can be specified.

- INSTR: searches by substring value, finds first match, returns position of beginning of substring. Both starting point and substring occurrence can be specified.

The following functions search for a substring by position or delimiter within a string and return the substring:

- $EXTRACT: searches by string position, returns substring specified by start position, or start and end positions. Searches from beginning of string.

- SUBSTRING: searches by string position, returns substring specified by start position, or start and length. Searches from beginning of string.

- SUBSTR: searches by string position, returns substring specified by start position, or start and length. Searches from beginning or end of string.

- $PIECE: searches by delimiter character, returns first delimited substring. Starting point can be specified or defaults to beginning of string.

- $LENGTH: searches by delimiter character, returns the number of delimited substrings. Searches from beginning of string.

- $LIST: searches by substring count on a specially encoded list string. It locates a substring by substring count and returns the substring value. Searches from beginning of string.

The contains operator ([) can also be used to determine if a substring appears in a string.

The %STARTSWITH comparison operator matches the specified character(s) against the beginning of a string.

Substring Search–and–Replace

The following functions search for a substring within a string and replace it with another substring.

- REPLACE: searches by string value, replaces substring with new substring. Searches from beginning of string.

- STUFF: searches by string position and length, replaces substring with new substring. Searches from beginning of string.

Character-Type and Word-Aware Comparisons

The %PATTERN comparison operator matches a string to a specified pattern of character types.

You can perform a word-aware search of a string for specified words or phrases, including wildcard searching. For further details refer to Using InterSystems SQL Search.