Examine Query Performance

Version 2024.1
2024-07-02

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Examine Query Performance
InterSystems IRIS Data Platform Version 2024.1 2024-07-02

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

1

Analyze SQL Statements and Statistics

This page discusses how to view and read SQL Statements and SQL Runtime Statistics to analyze query performance. Both tools are useful for understanding at a high level how well queries are performing on your system.

SQL Statements provide a record of SQL queries and other operations for each table, including insert, update, and delete. These statements are linked to a query plan, and this link provides the option to freeze this query plan. The system creates an SQL Statement for each SQL DML operation and stores them in a list, viewable in the Management Portal. If you change the table definition, you can use this list to determine whether the query plan for each SQL operation will be affected by this DML change or an SQL operation may need to be modified or both. You can then:

- Determine which query plan to use for each SQL operation. You can decide to use a revised query plan that reflects changes made to the table definition or you can freeze the current query plan, retaining the query plan generated prior to making changes to the table definition.
- Determine whether to make code changes to routines that perform SQL operations against that table, based on changes made to the table definition.

Note: SQL Statements are a listing of SQL routines that may be affected by a change to a table definition. It *should not* be used as a history of changes to either the table definition or table data.

SQL Runtime Statistics can track basic metrics of the execution of all SQL queries to provide you with a historical overview of the performance of your queries. When the collection of these statistics is turned on, the system will automatically collect metrics on the runtime performance of your queries until the collection process times out. Analyzing these statistics can provide insights into how your schemas are performing and determine which queries may need to be examined more thoroughly using the SQL Performance Analysis Toolkit.

1.1 Operations that Create SQL Statements

The following SQL operations create corresponding SQL Statements:

- Data management (DML) operations include queries against the table, and insert, update, and delete operations. Each data management (DML) operation (both Dynamic SQL and Embedded SQL) creates an SQL Statement when the operation is executed. Dynamic SQL SELECT commands create an entry in the Management Portal **Cached Queries** listing.
- Embedded SQL cursor-based SELECT commands create an SQL Statement when the OPEN command invokes a DECLARED query. No separate entry is created in the Management Portal **Cached Queries** listing.

If a query references more than one table, a single SQL Statement is created in the namespace's **SQL Statements** listing that lists all of the referenced tables in the Table/View/Procedure Name(s) column. The **Table's SQL Statements** listing for each referenced table contains an entry for that query.

An SQL Statement is created when the query is prepared for the first time. If more than one client issues the same query only the first prepare is recorded. For example, if JDBC issues a query and then ODBC issues an identical query, the SQL Statement index would only have information about the first JDBC client and not the ODBC client.

Most SQL Statements have an associated Query Plan. When created, this Query Plan is unfrozen; you can subsequently designate this Query Plan as a frozen plan. SQL Statements with a Query Plan include DML commands that involve a SELECT operation. Also see SQL Statements without a Query Plan.

Note: SQL Statements only list the most recent version of an SQL operation. Unless you freeze the SQL Statement, InterSystems IRIS® data platform replaces it with the next version. Thus rewriting and invoking the SQL code in a routine causes the old SQL code to disappear from SQL Statements.

1.1.1 Other SQL Statement Operations

The following SQL commands perform more complex SQL Statement operations:

- **CREATE TRIGGER**: No SQL Statement is created in the table in which the trigger is defined, either when the trigger is defined or when it is pulled. However, if the trigger performs a DML operation on another table, defining a trigger creates an SQL Statement for the table modified by the trigger code. The **Location** specifies the table in which the trigger is defined. The SQL Statement is defined when the trigger is defined; dropping a trigger deletes the SQL Statement. Pulling a trigger does not create an SQL Statement.
- **CREATE VIEW** does not create an SQL Statement, because nothing is compiled. It also does not change the Plan Timestamp of the SQL Statements of its source table. However, compiling a DML command for a view creates an SQL Statement for that view.

1.2 Listing SQL Statements

From the Management Portal SQL interface, you can list SQL Statements as follows:

- **SQL Statements** tab: this lists all SQL Statements in the namespace, in collation sequence by schema then by table/view name within each schema. This listing only includes those tables/views for which the current user has privileges. If a SQL Statement references more than one table, the Table/View/Procedure Name(s) column lists all the referenced tables in alphabetical order.

  By clicking a column heading you can sort the list of SQL Statements by **Table/View/Procedure Name(s)**, **Plan State**, **Location(s)**, **SQL Statement Text**, or any other column of the list. These sortable columns enable you to quickly find, for example, all frozen plans (**Plan State**), all cached queries (**Location(s)**), or the slowest queries (**Average time**).

  You can use the **Filter** option provided with this tab to narrow the listed SQL Statements to a specified subset. A specified filter string filters on all data in the SQL Statements listing, most usefully on schema or schema.table name, routine location, or a substring found in the SQL Statement texts. A filter string is not case-sensitive, but must follow statement text punctuation whitespace (`name , age`, not `name,age`). If a query references more than one table, the **Filter** includes the SQL Statement if it selects for any referenced table in the **Table/View/Procedure Name(s)** column. The **Filter** option is user customized.

  The **Max rows** option defaults to 1,000. The maximum value is 10,000. The minimum value is 10. To list more than 10,000 SQL Statements, use INFORMATION_SCHEMA.STATEMENTS. The **Page size** and **Max rows** options are user customized.

- **Catalog Details** tab: select a table and display its catalog details. This tab provides an **Table's SQL Statements** button to display the SQL Statements associated with that table. Note that if a SQL Statement references more than one table, it will be listed in the **Table's SQL Statements** listing for each referenced table, but only the currently selected table is listed in the **Table Name** column.

  By clicking a column heading you can sort the list of the table's SQL Statements by any column of the list.

You can use INFORMATION_SCHEMA.STATEMENTS to list SQL Statements selected by various criteria, as described in Querying the SQL Statements, below.

1.2.1 Listing Columns

The **SQL Statements** tab lists all SQL statements in the namespace. The **Catalog Details** tab **Table's SQL Statements** button lists the SQL Statements for the selected table. Both listings contain the following column headings:

- **#**: a sequential numbering of the list rows. These numbers are not associated with specific SQL Statements.
- **Table/View/Procedure Name(s)**: the qualified SQL table (or view or procedure) name(s). If a query references multiple tables or views:
  - The **SQL Statements** tab for the namespace lists all of the tables and views in **Table/View/Procedure Name(s)** in collation sequence (case-insensitive alphabetical order). To determine which are tables and which are views, select the SQL Statement to display the SQL Statement Details, Statement Uses the Following Relations.
  - The **Table's SQL Statements** lists all of the SQL Statements that reference that table. **Table/View/Procedure Name(s)** lists only the specified table. Therefore, the same SQL Statement can appear in the **Table's SQL Statements** listings for multiple tables.
- **Plan State**
- **New plan**
- **Execution count**
- **Execution count/day**
- **Total Time**
- **Average Time**
- **StdDev time**
- **Row Count**
- **Row Count/day**
- **Location(s)**: the location of the compiled query:
  - Dynamic SQL: the cached query name. For example `%sqlcq.USER.cls2.1`.
  - Embedded SQL: the routine name. For example `MyESQL`.
  - Stored Procedure: the class name for the stored procedure. For example `Sample.procNamesJoinSP.1`.
- **SQL Statement Text**: the SQL Statement text (truncated to 128 characters) in normalized format, which may differ from the command text, as specified in SQL Statement text below.

You can use the **Filter** option provided with this tab to filter by a **Location(s)** column value

1.2.2 Plan State

The **Plan State** lists one of the following:

- Unfrozen: not frozen, can be frozen.
- Unfrozen/Parallel: not frozen, cannot be frozen.
- Frozen/Explicit: frozen by user action, can be unfrozen.
- Frozen/Upgrade: frozen by InterSystems IRIS version upgrade, can be unfrozen.
- blank: no associated Query Plan:
  - An INSERT... VALUES() command creates an SQL Statement that does not have an associated Query Plan, and therefore cannot be unfrozen or frozen (the **Plan State** column is blank). Even though this SQL command does not produce a Query Plan, its listing in SQL Statements still is useful, because it allows you to quickly locate all the SQL operations against this table. For example, if you add a column to a table, you may want to find out where all of the SQL INSERTs are for that table so you can update these commands to include this new column.
  - A cursor-based UPDATE or DELETE command does not have an associated Query Plan, and therefore cannot be unfrozen or frozen (the **Plan State** column is blank). Executing the OPEN command for a declared CURSOR generates an SQL Statement with an associated Query Plan. Embedded SQL statements that use that cursor (**FETCH cursor**, **UPDATE...WHERE CURRENT OF cursor**, **DELETE...WHERE CURRENT OF cursor**, and **CLOSE cursor**) do not generate separate SQL Statements. Even though a cursor-based UPDATE or DELETE does not produce a Query Plan, its listing in SQL Statements is still useful, because it allows you to quickly locate all the SQL operations against this table.

1.2.3 SQL Statement Text

The SQL Statement text commonly differs from the SQL command because SQL statement generation normalizes lettercase and whitespace. Other differences are as follows:

- If you issue a query from the Management Portal interface or the SQL Shell interface, the resulting SQL Statement differs from the query by preceding the SELECT statement with `DECLARE QRS CURSOR FOR` (where "QRS" can be a variety of generated cursor names). This allows the statement text to match that of the Dynamic SQL cached query.
- If the SQL command specifies an unqualified table or view name, the resulting SQL Statement provides the schema by using either a schema search path (for DML, if provided) or the default schema name.
- SQL Statement Text is truncated after 1024 characters. To view the complete SQL Statement Text, display the SQL Statement Details.
- A single SQL command may result in more than one SQL Statement. For example, if a query references a view, **SQL Statements** displays two statement texts, one listed under the view name, the other listed under the underlying table name. Freezing either statement results in **Plan State** of Frozen for both statements.
- When SQL statements are prepared via a database driver, SQL statement generation appends SQL Comment Options (#OPTIONS) to the statement text if the options are needed to generate the statement index hash. This is shown in the following example:

```
DECLARE C CURSOR FOR SELECT * INTO :%col(1) , :%col(2) , :%col(3) , :%col(4) , :%col(5)
FROM SAMPLE . COMPANY /*#OPTIONS {"xDBCIsoLevel":0} */
```

1.2.4 Stale SQL Statements

When a routine or class associated with an SQL Statement is deleted, the SQL Statement listing is not automatically deleted. This type of SQL Statement listing is referred to as Stale. Since it is often useful to have access to this historic information and the usage statistics associated with the SQL Statement, these stale entries are preserved in the Management Portal SQL Statement listing.

You can remove these stale entries by using the **Clean Stale** button. **Clean Stale** removes all non-frozen SQL Statements for which the associated routine or class (table) is no longer present or no longer contains the SQL Statement query. **Clean Stale** does not remove frozen SQL Statements. You can perform the same clean stale operation using the **$SYSTEM.SQL.Statement.Clean()** method.

If you delete a table (persistent class) associated with an SQL Statement, the **Table/View/Procedure Name(s)** column is modified, as in the following example: `SAMPLE.MYTESTTABLE - Deleted??`; the name of the deleted table is converted to all uppercase letters and is flagged as "Deleted??". Or, if the SQL Statement referenced more than one table: `SAMPLE.MYTESTTABLE - Deleted?? Sample.Person`.

- For a Dynamic SQL query, when you delete the table the **Location(s)** column is blank because all cached queries associated with the table have been automatically purged. **Clean Stale** removes the SQL Statement.
- For an Embedded SQL query, the **Location(s)** column contains the name of the routine used to execute the query. When you change the routine so that it no longer executes the original query, the **Location(s)** column is blank. **Clean Stale** removes the SQL Statement. When you delete a table used by the query, the table is flagged as "Deleted??"; **Clean Stale** does not remove the SQL Statement.

Note: A system task is automatically run once per hour in all namespaces to clean up indexes for any SQL Statements that might be stale or have stale routine references. This operation is performed to maintain system performance. This internal clean-up is not reflected in the Management Portal SQL Statements listings.

1.3 Data Management (DML) SQL Statements

The Data Management Language (DML) commands that create an SQL Statements are: INSERT, UPDATE, INSERT OR UPDATE, DELETE, TRUNCATE TABLE, SELECT, and OPEN cursor for a declared cursor-based SELECT. You can use Dynamic SQL or Embedded SQL to invoke a DML command. A DML command can be invoked for a table or a view, and InterSystems IRIS creates a corresponding SQL Statement.

Note: The system creates an SQL Statement when Dynamic SQL is prepared or when an Embedded SQL cursor is opened, *not* when the DML command is executed. The SQL Statement timestamp records when this SQL code invocation occurred, not when (or if) the query was executed. Thus an SQL Statement may represent a change to table data that was never actually performed.

Preparing a Dynamic SQL DML command creates a corresponding SQL Statement. The **Location** associated with this SQL Statement is a cached query. Dynamic SQL is prepared when SQL is executed from the Management Portal SQL interface, from the SQL Shell interface, or imported from a .txt file. Purging an unfrozen cached query flags the corresponding SQL Statement for **Clean Stale** deletion. Purging a frozen cached query removes the **Location** value for the corresponding SQL Statement. Unfreezing the SQL Statement flags it for **Clean Stale** deletion.

Executing a non-cursor Embedded SQL Data Management Language (DML) command creates a corresponding SQL Statement. Each Embedded SQL DML command creates a corresponding SQL Statement. If a routine contains multiple Embedded SQL commands, each Embedded SQL command creates a separate SQL Statement. (Some Embedded SQL

commands create multiple SQL Statements.) The **Location** column of the SQL Statement listing specifies the routine that contains the Embedded SQL. In this way, SQL Statements maintains a record of each Embedded SQL DML command.

Opening a cursor-based Embedded SQL Data Management Language (DML) routine creates an SQL Statement with a Query Plan. Associated Embedded SQL statements (**FETCH cursor**, **CLOSE cursor**) do not generate separate SQL Statements. Following a **FETCH cursor**, an associated **UPDATE table WHERE CURRENT OF cursor** or **DELETE FROM table WHERE CURRENT OF cursor** does generate a separate SQL Statement, but no separate Query Plan.

An **INSERT** command that inserts literal values creates a SQL Statement with the **Plan State** column blank. Because this command does not create a Query Plan, the SQL Statement cannot be frozen.

1.3.1 SELECT Commands

Invoking a query creates a corresponding SQL Statement. It can be a simple SELECT, or a CURSOR-based SELECT/FETCH operation. The query can be issued against a table or a view. Some important details about these sorts of queries:

- A query containing a JOIN creates an identical SQL Statement for each table. The **Location** is the same stored query in the listing for each table. The **Statement Uses the Following Relations** lists all of the tables, as described in the SQL Statement Details Routines and Relations Sections.
- A query containing a selectItem subquery creates an identical SQL Statement for each table. The **Location** is the same stored query in the listing for each table. The **Statement Uses the Following Relations** lists all of the tables, as described in the SQL Statement Details Routines and Relations Sections.
- A query that references an external (linked) table cannot be frozen.
- A query containing the FROM clause %PARALLEL keyword may create more than one SQL Statement. You can display these generated SQL Statements by invoking:

  SQL

  ```
  SELECT * FROM INFORMATION_SCHEMA.STATEMENT_CHILDREN
  ```

  This displays the Statement column containing the statement hash of the original query and the ParentHash column containing the statement hash of a generated version of the query.

  SQL Statements for a %PARALLEL query have a Plan State of Unfrozen/Parallel, and cannot be frozen.
- A query containing no FROM clause, and therefore not referencing any table, still creates an SQL Statement. For example: `SELECT $LENGTH('this string')` creates a SQL Statement with the **Table** column value `%TSQL_sys.snf`.

1.4 SQL Statement Details

There are two ways to display the **SQL Statement Details**:

- From the **SQL Statements** tab, select an SQL Statement by clicking the **Table/View/Procedure Name(s)** link in the left-hand column. This displays the **SQL Statement Details** in a separate tab. This interface allows you to open multiple tabs for comparison. It also provides a **Query Test** button that displays the SQL Runtime Statistics page.
- From the table's **Catalog Details** tab (or the **SQL Statements** tab), select an SQL Statement by clicking the **Statement Text** link in the right-hand column. This displays the **SQL Statement Details** in a pop-up window.

You can use either **SQL Statement Details** display to view the Query Plan and to freeze or unfreeze the query plan.

SQL Statement Details provides buttons to **Freeze** or **Unfreeze** the query plan. It also provides a **Clear SQL Statistics** button to clear the usage statistics listed under Statement Details, an **Export** button to export one or more SQL Statements to a file, as well as a buttons to **Refresh** and to **Close** the page.

The **SQL Statement Details** display contains the following sections. Each of these sections can be expanded or collapsed by selecting the arrow icon next to the section title:

- Statement Details (includes Usage Statistics)
- Compile Settings
- Statement is Defined in the Following Routines
- Statement Uses the Following Relations
- Statement Text and Query Plan (described elsewhere)

1.4.1 Statement Details Section

Statement Details section:

- **Plan state**: Frozen/Explicit, Frozen/Upgrade, Unfrozen, or Unfrozen/Parallel. Frozen/Explicit means that this statement's plan has been frozen by an explicit user action and this frozen plan is the query plan that will be used, regardless of changes to the code that generated this SQL Statement. Frozen/Upgrade means that this statement's plan has been automatically frozen by an InterSystems IRIS version upgrade. Unfrozen means that the plan is currently unfrozen and may be frozen. Unfrozen/Parallel mean that the plan is unfrozen and uses %PARALLEL processing, and therefore cannot be frozen. A NULL (blank) plan state means that there is no associated query plan.
- **Total time**: the amount of time (in seconds) that running this query has taken.
- **Version**: the InterSystems IRIS version under which the plan was created. If the **Plan state** is Frozen/Upgrade, this is an earlier version of InterSystems IRIS. When you unfreeze a query plan, the **Plan state** is changed to Unfrozen and the **Version** is changed to the current InterSystems IRIS version.
- **Execution count**: an integer count of the number of times this query has been run. A change that results in a different Query Plan for this query (such as adding an index to a table) will reset this count.
- **Average time**: the average amount of time (in seconds) that running this query has taken. If the query is a cached query, the first execution of the query likely took significantly more time than subsequent executions of the optimized query from the query cache.
- **Row Count**: the total number of rows returned or modified by this query.
- **Date first seen**: the date the query was first run (executed). This may differ from the **Last Compile Time**, which is when the query was prepared.
- **Execution count/day**: the average number of times this query is run per day.
- **Stddev time**: the standard deviation of this query's total runtime from the average runtime. The standard deviation measures the variability of the time for this query.
- **Row count/day**: the average number of rows this query returns or modifies per day.
- **Timestamp**: Initially, the timestamp when the plan was created. This timestamp is updated following a freeze / unfreeze to record the time the plan was unfrozen, not the time the plan was re-compiled. You may have to click the **Refresh Page** button to display the unfreeze timestamp. Comparing the **Plan Timestamp** with the datetime value of the routine/class that contain the statement will let you know if the routine/class is not using the same query plan if it was recompiled again.

- **Frozen plan different**: if you freeze the plan, this additional field is displayed, displaying whether the frozen plan is different from the unfrozen plan. When you freeze the plan, the Statement Text and Query Plan displays the frozen plan and the unfrozen plan side-by-side for easier comparison.
- **Statement hash**: an internal hash representation of the statement definition that is used as the key of the SQL Statement Index (for internal use only). 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.

1.4.1.1 Usage Statistics

In the **Statement Details** section, these fields provide usage statistics for the SQL statement:

- **Total time**
- **Execution count**
- **Average time**
- **Date first seen**
- **Execution count/day**
- **Stddev time**
- **Row Count**
- **Row Count/day**

This information can be used to determine which queries are the slowest and which queries are executed the most. By using this information you can determine which queries would provide significant benefits by being optimized.

Query usage statistics are periodically updated for completed query executions. You can use the **Clear SQL Statistics** button to clear the values of these fields.

InterSystems IRIS does not separately record usage statistics for %PARALLEL subqueries. %PARALLEL subquery statistics are summed with the statistics for the outer query. Queries generated by the implementation to run in parallel do not have their usage statistics tracked individually.

You can view these query usage statistics for multiple SQL statements in the SQL Statements tab display. You can sort the **SQL Statements** tab listing by any column. This makes it easy to determine, for example, which queries have the largest average time.

You can also access these query usage statistics by querying the INFORMATION.SCHEMA.STATEMENTS class properties, as described in Querying the SQL Statements. In addition, the Management Portal's **SQL Activity** page provides these usage statistics for queries in progress.

1.4.2 Compile Settings Section

Compile Settings section:

- **Select mode**: the SelectMode the statement was compiled with. For DML commands this can be set using #sqlcompile select; the default is Logical. If #sqlcompile select=Runtime, a call to the SelectMode option of the **$SYSTEM.SQL.Util.SetOption()** method can change the query result set display, but does not change the **Select Mode** value, which remains Runtime.
- **Default schema(s)**: the default schema name that were set when the statement was compiled. This is commonly the default schema in effect when the command was issued, though SQL may have resolved the schema for unqualified names using a schema search path (if provided) rather than the default schema name. However, if the statement is a

DML command in Embedded SQL using one or more #import macro directives, the schemas specified by **#import** directives are listed here.

- **Schema path**: the schema path defined when the statement was compiled. This is the schema search path, if specified. If no schema search path is specified, this setting is blank. However, for a DML Embedded SQL command with a search path specified in an #import macro directive, the **#import** search path is shown in the **Default schema(s)** setting and this **Schema path** setting is blank.
- **Plan Error**: This field only appears when an error occurs when using a frozen plan. For example, if a query plan uses an index, the query plan is frozen, and then the index is dropped from the table, a **Plan Error** occurs such as the following: `Map 'NameIDX' not defined in table 'Sample.Person', but it was specified in the frozen plan for the query.` Dropping or adding an index causes a recompile of the table, changing the **Last Compile Time** value. The **Clear Error** button can be used to clear the **Plan Error** field once the condition that caused the error has been corrected — for example, by re-creating the missing index. Using the **Clear Error** button after the error condition has been corrected causes both the **Plan Error** field and the **Clear Error** button to disappear. For further details, refer to Frozen Plan in Error.

1.4.3 Routines and Relations Sections

Statement is Defined in the Following Routines section:

- **Routine**: the class name associated with the cached query (for Dynamic SQL DML), or the routine name (for Embedded SQL DML).
- **Type**: Class Method or MAC Routine (for Embedded SQL DML).
- **Last Compile Time**: the last compile time or prepare time for the routine. If the SQL Statement is Unfrozen, recompiling a MAC routine updates both this timestamp and the **Plan Timestamp**. If the SQL Statement is Frozen, recompiling a MAC routine updates only this timestamp; the **Plan Timestamp** is unchanged until you unfreeze the plan; the **Plan Timestamp** then shows the time the plan was unfrozen.

Statement Uses the Following Relations section lists one or more defined tables used to create the query plan. For an **INSERT** that uses a query to extract values from another table, or an **UPDATE** or **DELETE** that uses a FROM clause to reference another table, both tables are listed here. For each table the following values are listed:

- **Table or View Name**: the qualified name of the table or view.
- **Type**: Table or View.
- **Last Compile Time**: The time the table (persistent class) was last compiled.
- **Classname**: the classname associated with the table.

This section includes a **Compile Class** option to re-compile the class. If you re-compile an unfrozen plan, all three time fields are updated. If you re-compile a frozen plan, the two **Last Compile Time** fields are updated, but the **Plan Timestamp** is not. When you unfreeze the plan and click the **Refresh Page** button, the **Plan Timestamp** updates to the time the plan was unfrozen.

1.5 Querying SQL Statements

You can use the INFORMATION_SCHEMA package tables to query the list of SQL Statements. InterSystems IRIS supports the following classes:

- INFORMATION_SCHEMA.STATEMENTS: Contains SQL Statement Index entries that can be accessed by the current user in the current namespace.

- INFORMATION_SCHEMA.STATEMENT_LOCATIONS: Contains each routine location from which an SQL statement is invoked: the persistent class name or the cached query name.
- INFORMATION_SCHEMA.STATEMENT_RELATIONS: Contains each table or view entry use by an SQL statement.
- INFORMATION_SCHEMA.CURRENT_STATEMENTS: Contains SQL Statement Index entries which are currently executing in any namespace on the system. Users with the access to the **%Admin_Operate** resource can explore the contents of this table at any time via the Management Portal, on the **SQL Activity** page.

Some example queries that use these classes are as follows:

1. **SQL**

```
SELECT Hash,Frozen,Timestamp,Statement
FROM INFORMATION_SCHEMA.STATEMENTS
```

   This example returns all of the SQL Statements in the namespace, listing the hash value (a computed Id that uniquely identifies the normalized SQL statement), the frozen status flag (values 0 through 3), the local timestamp when the statement was prepared and the plan saved, and the statement text itself.

2. **SQL**

```
SELECT Frozen,FrozenDifferent,Timestamp,Statement
FROM INFORMATION_SCHEMA.STATEMENTS
WHERE Frozen=1 OR Frozen=2
```

   This example returns the SQL Statements for all frozen plans, indicating whether the frozen plan is different from what the plan would be if not frozen. Note that an unfrozen statement may be Frozen=0 or Frozen=3. A statement such as a single row INSERT, that cannot be frozen, displays NULL in the Frozen column.

3. **SQL**

```
SELECT Statement,Frozen,
  STATEMENT_LOCATIONS->Location AS Routine,STATEMENT_LOCATIONS->Type AS RoutineType
FROM INFORMATION_SCHEMA.STATEMENTS
WHERE STATEMENT_RELATIONS->Relation='SAMPLE.PERSON'
```

   This example returns all the SQL Statements and the routines the statements are located in for a given SQL table. (Note that the table name (`SAMPLE.PERSON`) must be specified with the same letter case used in the SQL Statement text: all uppercase letters).

4. **SQL**

```
SELECT Statement,Frozen,Frozen_Different,
  STATEMENT_LOCATIONS->Location AS Routine,STATEMENT_LOCATIONS->Type AS RoutineType
FROM INFORMATION_SCHEMA.STATEMENTS
WHERE Frozen=1 OR Frozen=2
```

   This example returns all the SQL Statements in the current namespace that have frozen plans.

5. **SQL**

```
SELECT Statement,Frozen,
  STATEMENT_LOCATIONS->Location AS Routine,STATEMENT_LOCATIONS->Type AS RoutineType
FROM INFORMATION_SCHEMA.STATEMENTS
WHERE Statement [ ' COUNT ( * ) '
```

   This example returns all the SQL Statements in the current namespace that contain a **COUNT(*)** aggregate function. Note that the statement text (`COUNT ( * )`) must be specified with the same whitespace used in the SQL Statement text.

1.6 Exporting and Importing SQL Statements

You can export or import SQL Statements as an XML-formatted text file. This enables you to move a frozen plan from one location to another. SQL Statement exports and imports include the associated query plan.

You can export a single SQL Statement or export all of the SQL Statements in the namespace.

You can import a previously-exported XML file containing one or more SQL Statements.

Note: This import of SQL Statements as XML should not be confused with the import and execution of SQL DDL code from a text file.

1.6.1 Exporting SQL Statements

Export a single SQL Statement:

- Use the SQL Statement Details page **Export** button. From the Management Portal **System Explorer** SQL interface, select the **SQL Statements** tab and click on a statement to open up the **SQL Statement Details** page. Select the **Export** button. This opens a dialog box, allowing you to select to export the file to Server (a data file) or Browser.
  - Server (the default): Enter the full path name of the export xml file. The first time you export, this file has a default name of `statementexport.xml`. You can, of course, specify a different path and file name. After you have successfully exported an SQL Statement file, the last used file name becomes the default.

    The **Run export in the background** check box is not selected by default.
  - Browser: Exports the file `statementexport.xml` to a new page in the user's default browser. You can specify another name for the browser export file, or specify a different software display option.
- Use the **$SYSTEM.SQL.Statement.ExportFrozenPlans()** method.

Export all SQL Statements in the namespace:

- Use the **Export All Statements** Action from the Management Portal. From the Management Portal **System Explorer** SQL interface, select the Actions drop-down list. From that list select **Export All Statements**. This opens a dialog box, allowing you to export all SQL Statements in the namespace to Server (a data file) or Browser.
  - Server (the default): Enter the full path name of the export xml file. The first time you export, this file has a default name of `statementexport.xml`. You can, of course, specify a different path and file name. After you have successfully exported an SQL Statement file, the last used file name becomes the default.

    The **Run export in the background** check box is selected by default. This is the recommended setting when exporting all SQL Statements. When **Run export in the background** is checked, you are provided with a link to view the background list page where you can see the background job status.
  - Browser: Exports the file `statementexport.xml` to a new page in the user's default browser. You can specify another name for the browser export file, or specify a different software display option.
- Use the **$SYSTEM.SQL.Statement.ExportAllFrozenPlans()** method.

1.6.2 Importing SQL Statements

Import an SQL Statement or multiple SQL Statements from a previously-exported file:

- Use the **Import Statements** Action from the Management Portal. From the Management Portal **System Explorer** SQL interface, select the Actions drop-down list. From that list select **Import Statements**. This opens a dialog box, allowing you to specify the full path name of the import XML file.

  The **Run import in the background** check box is selected by default. This is the recommended setting when importing a file of SQL Statements. When **Run import in the background** is checked, you are provided with a link to view the background list page where you can see the background job status.

- Use the **$SYSTEM.SQL.Statement.ImportFrozenPlans()** method.

1.6.3 Viewing and Purging Background Tasks

From the Management Portal **System Operation** option, select **Background Tasks** to view the log of export and import background tasks. You can use the **Purge Log** button to clear this log.

1.7 SQL Runtime Statistics

You can use SQL Runtime Statistics to monitor general performance statistics, such as query runtime or commands executed, of **SELECT** queries, as well as DDL and DML statements running on your system. SQL runtime statistics are gathered when a query operation is prepared and are viewable from the INFORMATION_SCHEMA.STATEMENTS, STATEMENT_DAILY_STATS, and STATEMENT_HOURLY_STATS tables. In addition, if parameter sampling is enabled, you can view statistics collected about runtime parameters in the INFORMATION_SCHEMA.STATEMENT_PARAMETER_STATS table.

The gathering of SQL runtime statistics is always on and cannot be turned off. To ensure that the gathering of statistics is as efficient as possible, the statistics are only written at set intervals. As a result, it may take up to 30 minutes to see gathered runtime statistics. To see which statements are currently running, query the INFORMATION_SCHEMA.CURRENT_STATEMENTS table.

Runtime statistics include:

- **Avg Time**: the average length of time the query takes in seconds
- **Run Count**: the number of times the query has been run
- **Avg Rows**: the average number of rows returned
- **Avg Commands**: the average number of commands executed

You can explicitly purge (clear) SQL runtime statistics. Purging a cached query deletes any related SQL runtime statistics. Dropping a table or view deletes any related SQL runtime statistics.

Note: A system task is automatically run once per hour in all namespaces to aggregate process-specific SQL query statistics into global statistics. Therefore, the global statistics may not reflect statistics gathered within the hour.

1.7.1 Runtime Statistics and Show Plan

The **SQL Runtime Statistics** tool can be used to display the Show Plan for a query with runtime statistics.

The **Alternate Show Plans** tool can be used to compare show plans with statistics, displaying runtime statistics for a query. The Alternate Show Plans tool in its **Show Plan Options** displays estimated statistics for a query. If gathering runtime statistics is activated, its **Compare Show Plans with Stats** option displays actual runtime statistics; if runtime statistics are not active, this option displays estimate statistics.

2

Interpreting an SQL Query Plan

This page explains how to read a system-generated InterSystems SQL Query Plan. It breaks down the tools you can use to view these plans, as well as how to interpret the language you will find in them.

When a SQL query is compiled, the process produces a set of instructions to access and return the data specified by the query. The instructions and the sequence in which they are executed are influenced by the data the SQL compiler has about the structure and content of the tables involved in the query. The compiler attempts to use information such as table sizes and available indexes to make the set of instructions as efficient as possible.

The query access plan (Show Plan) is a human-readable translation of that resulting set of instructions. The author of the query can use this query access plan to see how the data will be accessed. While the SQL compiler tries to make the most efficient use of data as specified by the query, you may know more about some aspect of the stored data than is evident to the compiler. In this case, you can make use of the query plan to modify the original query to provide more information or more guidance to the query compiler.

2.1 Viewing the Plan

You can use the EXPLAIN or Show Plan tools to display an execution plan for **SELECT**, **DECLARE**, **UPDATE**, **DELETE**, **TRUNCATE TABLE**, and some **INSERT** operations. These are collectively known as query operations because they use a **SELECT** query as part of their execution. InterSystems IRIS generates an execution plan when a query operation is prepared; you do not have to actually execute the query to generate an execution plan.

By default, these tools display what InterSystems IRIS considers to be the optimal query plan. For most queries there is more than one possible query plan. In addition to the query plan that InterSystems IRIS deems as optimal, you can also generate and display alternate query execution plans.

InterSystems IRIS provides the following query plan tools:

- The SQL EXPLAIN command can be used to generate an XML-formatted query plan and, optionally, alternate query plans and SQL statistics. All generated query plans and statistics are included in a single result set field named Plan. Note that the **EXPLAIN** command can only be used with a SELECT query.
- The Management Portal—>**System Explorer**—>**SQL** interface **Show Plan** button.
- The Management Portal—>**System Explorer**—>**Tools**—>**SQL Performance Tools**.
- The $SYSTEM.SQL.Explain() method can be used to generate and display an XML-formatted query plan and, optionally, alternate query plans.
- The **SHOW PLAN** and **SHOW PLANALT** Shell commands can be used from the SQL Shell to display the execution plan for the most recently executed query.

For generated %PARALLEL and Sharded queries, these tools display all of the applicable query plans.

2.1.1 Using EXPLAIN

You can generate a query execution plan by executing an EXPLAIN command, like the one in the following example:

SQL

```
EXPLAIN SELECT TOP 10 Name,DOB FROM Sample.Person
```

To use the EXPLAIN command, a user must have the %Development:USE resource.

If you specify the ALT keyword, the EXPLAIN command will generate alternate query plans and include them in the returned query plan.

If you specify the STAT keyword, the EXPLAIN command will generate performance statistics for each module of the query. For each module, the following statistics are returned:

- <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.

The query plan is returned as an XML-formatted string. The highest order tag is <plans>. If you did not specify the ALT keyword, the <plans> tag will contain a single <plan> tag that has <sql> tags (which specify the query), a <cost> tag (which contains the relative cost of this particular plan), and some text that describes how the SQL optimizer processed the query. If you did specify the ALT keyword, multiple <plan> tags will appear in the <plans> tag. If you specified the STAT keyword, a series of <stats> tags will appear after the <cost> tag; a separate <stats> tag is generated for each module involved in the processing of the query.

2.1.2 Using Show Plan in the Management Portal

You can use Show Plan to display the execution plan for a query in any of the following ways:

- From the Management Portal **SQL** interface: Select **System Explorer**, then **SQL**. Select a namespace by clicking the name of the current namespace displayed at the top of the page. (You can set the Management Portal default namespace for each user.) Write a query, then press the **Show Plan** button. (You can also invoke Show Plan from the **Show History** listing by clicking the plan option for a listed query.) See Executing SQL Statements.
- From the Management Portal **Tools** interface: Select **System Explorer**, then **Tools**, then select **SQL Performance Tools**, then **SQL Runtime Statistics**:
  - From the **Query Test** tab: Select a namespace by clicking the name of the current namespace displayed at the top of the page. Write a query in the text box. Then press the **Show Plan with SQL Stats** button. This generates a Show Plan without executing the query.
  - From the **View Stats** tab: Press the **Show Plan** button for one of the listed queries. The listed queries include both those written at **Execute Query**, and those written at **Query Test**.

Show Plan by default returns values in Logical mode. However, when invoking Show Plan from the Management Portal or the SQL Shell, Show Plan uses Runtime mode.

2.2 Structure of the Plan

The Show Plan execution plan consists of two components, **Statement Text** and **Query Plan**.

Statement Text replicates the original query, with the following modifications: The **Show Plan** button from the Management Portal **SQL** interface displays the SQL statement with comments and line breaks removed. Whitespace is standardized. The **Show Plan** button display also performs literal substitution, replacing each literal with a ?, unless you have suppressed literal substitution by enclosing the literal value in double parentheses. These modifications are not performed when displaying a show plan using the **Explain()** method, or when displayed using the SQL Runtime Statistics or Alternate Show Plans tools.

Query Plan shows the plan that would be used to execute the query. A Query Plan can include the following:

- "Frozen Plan" is the first line of **Query Plan** if the query plan has been frozen; otherwise, the first line is blank.
- "Relative cost" is an integer value which is computed from many factors as an abstract number for comparing the efficiency of different execution plans for the same query. This calculation takes into account (among other factors) the complexity of the query, the presence of indexes, and the size of the table(s). Relative cost is not useful for comparing two different queries. "Relative cost not available" is returned by certain aggregate queries, such as COUNT(*) or MAX(%ID) without a WHERE clause.
- The **Query Plan** consists of a main module, and (when needed) one or more subcomponents. One or more module subcomponents may be shown, named alphabetically, starting with B: `Module:B`, `Module:C`, etc.), and listed in the order of execution (not necessarily alphabetically).

  A named subquery module is shown for each subquery in the query. Subquery modules are named alphabetically. Subquery naming skips one or more letters before each named subquery. When the end of the alphabet is reached, additional subqueries are numbered, parsing Z=26 and using the same skip sequence. The following example is an every-third subquery naming sequence starting with `Subquery:F`: F, I, L, O, R, U, X, 27, 30, 33. The following example is an every-second subquery naming sequence starting with `Subquery:G`: G, I, K, M, O, Q, S, U, W, Y, 27, 29. If a subquery calls a module, the module is placed in alphabetical sequence after the subquery with no skip. For example, `Subquery:H` might call `Module:I`.
- "Read master map" as the first bullet item in the main module indicates an inefficient Query Plan. The Query Plan begins execution with one of the following map type statements `Read master map...` (no available index), `Read index map...` (use available index), or `Generate a stream of idkey values using the multi-index combination...` (Multi Index, use multiple indexes). Because the master map reads the data itself, rather than an index to the data, `Read master map...` almost always indicates an inefficient Query Plan. Unless the table is relatively small, you should define an index so that when you regenerate the Query Plan the first map says `Read index map...`. For information on interpreting a Query Plan, refer to Interpret an SQL Execution Plan.

Some operations create a Show Plan that indicates no Query Plan could be generated:

- Non-query INSERT: An **INSERT... VALUES()** command does not perform a query, and therefore does not generate a Query Plan.
- Query always FALSE: In a few cases, InterSystems IRIS can determine when preparing a query that a query condition will always be false, and thus cannot return data. The Show Plan informs you of this situation in the Query Plan component. For example, a query containing the condition `WHERE %ID IS NULL` or the condition `WHERE Name %STARTSWITH('A') AND Name IS NULL` cannot return data, and therefore InterSystems IRIS generates no execution plan. Rather than generating an execution plan, the Query Plan says "`Output no rows`". If a query contains a subquery with one of these conditions, the subquery module of the Query Plan says "`Subquery result NULL,`

`found no rows`". This condition check is limited to a few situations involving NULL, and is not intended to catch all self-contradictory query conditions.

- Invalid query: Show Plan displays an SQLCODE error message for most invalid queries. However, in a few cases, Show Plan displays as empty. For example, `WHERE Name = $$$$$` or `WHERE Name %STARTSWITH('A")` (note single-quote and double-quote). In these cases, Show Plan displays no **Statement Text**, and **Query Plan** says `[No plan created for this statement]`. This commonly occurs when quotation marks delimiting a literal are imbalanced. It also occurs when you specify two or more leading dollar signs without specifying the correct syntax for a user-defined ("extrinsic") function.

2.3 Reading the Plan

The result of Show Plan is a series of statements about what processing will be done to access and present the data specified in the query. The following provides information on how to interpret Show Plan statements.

2.3.1 Accessing Maps

An SQL table is stored as a set of `maps`. Each table has a master map that contains all the data in the table; the table may also have other maps such as index maps and bitmaps. Each map can be pictured as a multidimensional global, with the data for some fields in one or more `subscripts`, and with the remaining fields stored in the node value. The subscripts control what data is being accessed.

- For the master map, the RowID or the *IDKEY* fields are normally used as the map subscripts.
- For an index map, normally other fields are used as the leading subscript(s), with the RowID/*IDKEY* fields as additional lower-level subscripts.
- For a bitmap, the bitmap layer can be thought of as an additional RowID subscript level. However, bitmaps can only be used for RowIDs that are positive integers.

The plan for a query could access several tables. When accessing a table, the plan may access a single map (index or master map), two maps (an index map followed by the master map), or, in the case of a `multi-index plan`, several maps.

In accessing the data via a map, the plan indicates the subscripts used. It also indicates what the actual subscript values will be: a single given value, a set of given values, a range of values, or all values present in the table for that subscript. Which one is chosen depends on the conditions specified in the query. Obviously, accessing a single subscript value or only a few subscript values is faster than accessing all the values at that subscript level.

2.3.2 Conditions and Expressions

When the query is run, various conditions specified by the query are tested. Except for certain subscript-limiting conditions as just mentioned, the Show Plan output does not explicitly indicate the testing of conditions. It is always best to test conditions as early as possible. The optimal place for testing the various conditions can be inferred from the plan details.

Similarly, Show Plan does not detail the computation of expressions and sub-expressions. Besides simplicity, the main reason for this is that in most database environments, table and index access constitute the more important aspect of processing; the cost of retrieving the table data dominates the overall query cost, as disk access speed is still orders of magnitude slower than CPU processing.

2.3.3 Loops

When accessing data from a table, it is often necessary to examine multiple rows iteratively. Such access is indicated by a `loop`. The instructions to be executed for each pass are referred to as the `body` of the loop. They are visually indicated by being indented. It is common for database access involving multiple tables to require loops within loops. In this case, each loop level is indicated by a further indentation when compared to the previous level.

2.3.4 Temporary Files

2.3.4.1 Definition

A query plan might also indicate the need to build and use an intermediate temporary file (`temp-file`). This is a "scratch" area in a local array. It is used to save temporary results for various purposes, such as sorting. Just like a map, a temp-file has one or more subscripts, and possibly also node data.

2.3.4.2 Use

Some temp-files contain data from processing a single table. In this instance, building the temp-file could be considered `pre-processing` for the data in that table. Reading such a temp-file may or may not be followed by accessing the master map of the source table. In other cases, temp-files could contain the results of processing multiple tables. In still other situations, temp-files are used to store grouped aggregate values, to check DISTINCT, etc.

You may notice that a query plan indicates that the query adds empty nodes to a temp file. This suggests that the query anticipates needing to save temporary results, but does not have any meaningful results to store in practice. If you see this behavior in the query plan, consider adding an index on your data; this can circumvent the creation of unused temp files.

2.3.5 Modules

The building of temp-files, as well as other processing, may be delegated to a separate unit of work called a `module`. Each module is named. When separate modules are listed, the plan indicates where each module is invoked. When execution of the module finishes, processing resumes at the next statement following the module invocation.

2.3.6 Queries Sent for Processing

For external tables linked through an ODBC or JDBC gateway connection, the plan shows the text of the query being sent to the remote SQL Gateway Connection to retrieve the requested data from the remote tables.

For parallel query processing and for sharding, the plan shows the various queries being sent to be processed in parallel or on the shards. The plan used for each of these queries is also displayed.

2.3.7 Sub-Queries, JOINs and UNIONs

Some subqueries (and views) within the given query might also be processed separately. Their plans are specified in separate `subquery` sections. The precise place where a subquery section is called from is not indicated in the plan. This is because they are often invoked as part of the processing of conditions or expressions.

For queries that specify OUTER JOIN, the plan may indicate the possible generation of a row of NULLs if no matching rows were found, in order to satisfy the requirements of the outer join semantics.

For UNION, the plan might indicate the combining of the result rows from the various union subqueries in a separate module, where further processing of these result rows may be done.

2.4 Alternate Show Plans

You can display alternate execution plans for a query using the Management Portal or the **Explain()** method.

To display alternate execution plans for a query from the Management Portal using either of the following:

- Select **System Explorer**, select **Tools**, select **SQL Performance Tools**, then select **Alternate Show Plans**.
- Select **System Explorer**, select **SQL**, then from the **Tools** drop-down menu select **Alternate Show Plans**.

Using the **Alternate Show Plans** tool:

1. Input an SQL query text, or retrieve one using the **Show History** button. You can clear the query text field by clicking the round "**X**" circle on the right hand side.
2. Press the **Show Plan Options** button to display multiple alternate show plans. The **Run ... in the background** check box is unselected by default, which is the preferred setting for most queries. It is recommended that you select the **Run ... in the background** check box for large or complex queries. While a long query is being run in background a **View Process** button is shown. Clicking **View Process** opens the Process Details page in a new tab. From the Process Details page, you can view the process, and may Suspend, Resume or Terminate the process.
3. Possible Plans are listed in ascending order by Cost, with the Map Type and Starting Map. You can select the **Show Plan** (no statistics) or **Show Plan with Stats** link for each plan for further details.
4. From the list of possible plans, use the check boxes to select the plans that you wish to compare, then press the **Compare Show Plans with Stats** button to run them and display their SQL statistics.

The **Explain()** method with the `all` qualifier shows all of the execution plans for a query. It first shows the plan the InterSystems IRIS considers optimal (lowest cost), then displays alternate plans. Alternate plans are listed in ascending order of cost.

The following example displays the optimal execution plan, then lists alternate plans:

ObjectScript

```
DO $SYSTEM.SQL.SetSQLStatsFlagJob(3)
SET mysql=1
SET mysql(1)="SELECT TOP 4 Name,DOB FROM Sample.Person ORDER BY Age"
DO $SYSTEM.SQL.Explain(.mysql,{"all":1},,.plan)
ZWRITE plan
```

Also refer to the **possiblePlans** methods in the %SYS.PTools.StatsSQL class.

2.4.1 Stats

The Show Plans Options lists assigns each alternate show plan a **Cost** value, which enables you to make relative comparisons between the execution plans.

The Alternate Show Plan details provides for each Query Plan a set of stats (statistics) for the Query Totals, and (where applicable) for each Query plan module. The stats for each module include Time (overall performance, in seconds), Global Refs (number of global references), Commands (number of lines executed), and Read Latency (disk wait, in milliseconds). The Query Totals stats also includes the number of Rows Returned.

3

SQL Performance Analysis Toolkit

InterSystems IRIS provides analysis tools that can be used to actively profile specific SQL statements. These tools gather detailed information about the execution of these SQL statements and are useful for pinpointing specific problems within a query plan. Using this information, developers can take steps to improve the performance of inefficient SQL statements. However, the active profiling can significantly increase the load on the server. Therefore, the SQL Performance Analysis Toolkit is meant for a concerted code analysis effort after examining the SQL Runtime Statistics and determining which specific queries need a closer look. It is not intended for continuous monitoring of executing code.

Note: %SYSTEM.SQL.PTools class methods are the preferred APIs for invoking this functionality. The %SYSTEM.SQL.PTools methods interface regroups and reorganizes functionality implemented in methods of the %SYS.PTools base classes.

3.1 Activate SQL Performance Statistics

The SQL Performance Analysis Toolkit offers support specialists the ability to profile specific SQL statements or groups of statements. By using these tools during the execution of specific SQL statements, you can gather performance statistics that can be used to analyze problematic statements in isolation or across a live workload.

Using methods in the %SYSTEM.SQL.PTools class, you can initiate the collection of advanced performance statistics. The following methods are provided to gather such statistics at different scales, including:

- The entire system: **setSQLStatsFlag()**
- A specific namespace: **setSQLStatsFlagByNS()**
- The current process or job: **setSQLStatsFlagJob()**
- A specified process or job: **setSQLStatsFlagByPID()**. If the first parameter is unspecified, or specified as $JOB or as an empty string (""), **setSQLStatsFlagJob()** is invoked. Thus `SET SQLStatsFlag=$SYSTEM.SQL.SetSQLStatsFlagByPID($JOB,3)` is equivalent to `SET SQLStatsFlag=$SYSTEM.SQL.SetSQLStatsFlagJob(3)`.

These methods take an integer action option. They return a colon-separated string, the first element of which is the prior action option. You can determine the current settings using the **getSQLStatsFlag()** or **getSQLStatsFlagByPID()** method.

You can invoke these method from ObjectScript or from SQL as shown in the following examples:

- from ObjectScript: `SET rtn=##class(%SYSTEM.SQL.PTools).setSQLStatsFlag(2,,8)`
- from SQL: `SELECT %SYSTEM_SQL.PTools_setSQLStatsFlag(2,,8)`

In addition, you can use the %PROFILE keyword (equivalent to `setSQLStatsFlagJob(2)`) or %PROFILE_ALL keyword (equivalent to `setSQLStatsFlagJob(3)`) in a SELECT, INSERT, UPDATE, or DELETE statement to gather performance analysis statistics for just that statement.

3.1.1 Action Option

For **setSQLStatsFlag()** and **setSQLStatsFlagByNS()** you specify one of the following Action options: 0 turn off statistics code generation; 1 turn on statistics code generation for all queries, but do not gather statistics (the default); 2 record statistics for just the outer loop of the query (gather statistics at the open and close of the query); 3 record statistics for all module levels of the query. Modules can be nested. If so, the MAIN module statistics are inclusive numbers, the overall results for the full query.

For **setSQLStatsFlagJob()** and **setSQLStatsFlagByPID()** the Action options differ slightly. They are: -1 turn off statistics for this job; 0 use the system setting value. The 1, 2, and 3 options are the same as **setSQLStatsFlag()** and override the system setting. The default is 0.

To gather SQL performance statistics, queries need to be compiled (Prepared) with statistics code generation turned on (option 1, the default):

- To go from 0 to 1: after changing the action option, Routines and Classes that contain SQL will need to be compiled to perform statistics code generation. You must purge cached queries to force code regeneration when working with Dynamic SQL or a database driver.
- To go from 1 to 2: you simply change the action option to begin gathering statistics. This allows you to enable SQL performance analysis on a running production environment with minimal disruption.
- To go from 1 to 3 (or 2 to 3): after changing the action option, Routines and Classes that contain SQL will need to be compiled to record statistics for all module levels. When working in Dynamic SQL or with a database driver, you must purge cached queries to force code regeneration. Option 3 is commonly only used on an identified poorly-performing query in a non-production environment.
- To go from 1, 2, or 3 to 0: to turn off statistics code generation you do not need to purge cached queries.

3.1.2 Collect Option

If the Action option is 2 or 3, when you invoke one of these methods you can specify a Collect option value to specify which performance statistics to collect. The default is to collect all statistics.

You specify a Collect option by adding together the integer values associated with each type of statistic that you wish to collect. The default is 15 (1 + 2 + 4 + 8).

These methods return the prior value of this Collect option as the second colon-separated element. You can determine the current setting using the **getSQLStatsFlag()** or **getSQLStatsFlagByPID()** method. By default all statistics are collected, returning 15 as the second element value.

Refer to %SYSTEM.SQL.PTools for further details.

3.1.3 Terminate Option

Statistics collection continues until terminated. By default, collection continues indefinitely until it is terminated by issuing another **setSQLStatsFlag[nnn]()** method. Or, if the Action option is 1, 2, or 3, you can specify a **setSQLStatsFlag[nnn]()** terminate option, either an elapsed period (in minutes) or a specified timestamp. You then specify the Action option re-set when that period elapses. For example, the string `"M:120:1"` sets M (elapsed minutes) to 120 minutes, at the end of which the Action option resets to 1. All other options reset to the default values appropriate for that Action option.

These methods return the prior value of this Terminate option value as the fifth colon-separated element as an encoded value. See Get Statistics Settings.

3.1.4 Activating Performance Statistics in the Management Portal

You can set the action option for collecting performance statistics in the Management Portal from the **Settings** tab on the SQL Runtime Statistics page by following either of the following paths:

- **System Explorer** > **Tools** > **SQL Performance Tools** > **SQL Runtime Statistics**
- **System Explorer** > **SQL** > **Tools** > **SQL Runtime Statistics**

3.2 Get Statistics Settings

The **setSQLStatsFlag[nnn]()** methods return the prior statistics settings as a colon-separated value. You can determine the current settings using the **getSQLStatsFlag()** or **getSQLStatsFlagByPID()** method.

The 1st colon-separated value is the Action option setting. The 2nd colon-separated value is the Collect option. The 3rd and 4th colon-separated values are used for namespace-specific statistics gathering. The 5th colon-separated value encodes the Terminate option. The 6th colon-separated value specifies the **FlagType**: 0=System flag, 1=Process/Job flag.

You can use the **ptInfo** array to display the Terminate option settings in greater detail, as shown in the following example:

ObjectScript

```
  KILL
  DO ##class(%SYSTEM.SQL.PTools).clearSQLStatsALL("USER")
  DO ##class(%SYSTEM.SQL.PTools).setSQLStatsFlagByNS("USER",3,,7,"M:5:1")
DisplaySettings
  SET SQLStatsFlag = ##class(%SYSTEM.SQL.PTools).getSQLStatsFlag(0,.ptInfo)
  WRITE "ptInfo array of performance statistics return value:",!
  ZWRITE ptInfo,SQLStatsFlag
```

3.3 Export Query Performance Statistics

You can export query performance statistics to a file using the **exportSQLStats()** method of %SYSTEM.SQL.PTools. This method is used to export statistics data from %SYSTEM.SQL.PTools classes to a file.

You can invoke **exportSQLStats()** as shown in the following examples:

- from ObjectScript: `SET status=##class(%SYSTEM.SQL.PTools).exportSQLStats("$IO")` (defaults to *format* T).
- from SQL: `CALL %SYSTEM_SQL.PTools_exportSQLStats('$IO')` (defaults to *format* H).

If you don't specify a *filename* argument, this method exports to the current directory. By default, this file is named `PT_StatsSQL_exportSQLStats_` followed by the current local date and time as YYYYMMDD_HHMMSS. You can specify $IO to output the data to the Terminal or Management Portal display. If you specify a filename argument, this method creates a file in the `Mgr` subdirectory for the current namespace, or in the path location you specify. This export is limited to data in the current namespace.

You can specify the output file *format* as P (text), D (comma-separated data), X (XML markup), H (HTML markup), or Z (user-defined delimiter).

By default this method exports the query performance statistics. You can specify that it instead export the SQL query text or the SQL Query Plan data, as shown in the following examples:

- Query Text: `CALL %SYSTEM_SQL.PTools_exportSQLStats('$IO',,0,1,0)`
- Query Plan: `CALL %SYSTEM_SQL.PTools_exportSQLStats('$IO',,0,1,1)`

exportSQLStats() modifies the query text by stripping out comments and performing literal substitution.

The same query text and query plan data can be returned by **ExportSQLQuery()**.

3.3.1 Stats Values

The following statistics are returned:

- RowCount - The total number of rows returned in the MAIN module for the given query.
- RunCount - The total number of times the query has been run since the last time it was compiled or prepared.
- ModuleCount - The total number of times a given module was entered during the run of the query.
- TimeToFirstRow - The total time spent to return the first resultset row to the MAIN module for the given query.
- TimeSpent - The total time spent in a given module for the given query.
- GlobalRefs - The total number of global references done in a given module for the given query.
- LinesOfCode - The total number of lines of ObjectScript code executed in a given module for the given query.
- DiskWait (also known as Disk Latency) - The total number of milliseconds spent waiting for disk reads in a given module for the given query.

SQL statistics report the aggregate counter values for all components of the query being run. Such components include all partitions of a parallel query or all shards of a sharded query.

3.4 Delete Query Performance Statistics

You can use the **clearSQLStatsALL()** method to delete performance statistics. By default, it deletes statistics gathered for all routines in the current namespace. You can either specify a different namespace or limit deletion to a specific routine or both.

Additionally, you can use the **Purge Stats** button from the **SQL Runtime Statistics** page in the Management Portal to delete all of the accumulated statistics for all queries in the current namespace. If successful, a message indicates the number of stats purged. If there were no stats, the `Nothing to purge` message is displayed. If the purge was unsuccessful, an error message is displayed.

3.5 Performance Statistics Examples

The following example gathers performance statistics on the main module of a query (Action option 2) that was prepared by the current process, then uses the **exportSQLStats()** to display the performance statistics to the Terminal.

ObjectScript

```
  DO ##class(%SYSTEM.SQL.PTools).clearSQLStatsALL()
  DO ##class(%SYSTEM.SQL.PTools).setSQLStatsFlagJob(2)
  SET myquery = "SELECT TOP 5 Name,DOB FROM Sample.Person"
  SET tStatement = ##class(%SQL.Statement).%New()
  SET qStatus = tStatement.%Prepare(myquery)
    IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET pStatus = ##class(%SYSTEM.SQL.PTools).exportSQLStats("$IO")
    IF pStatus'=1 {WRITE "Performance stats display failed:"
       DO $System.Status.DisplayError(qStatus) QUIT}
```

The following example gathers performance statistics on all modules of a query (Action option 3) that was prepared by the current process, then calls **exportSQLStats()** from Embedded SQL to display the performance statistics to the Terminal:

ObjectScript

```
  DO ##class(%SYSTEM.SQL.PTools).clearSQLStatsALL()
  DO ##class(%SYSTEM.SQL.PTools).setSQLStatsFlagJob(3)
  SET myquery = "SELECT TOP 5 Name,DOB FROM Sample.Person"
  SET tStatement = ##class(%SQL.Statement).%New()
  SET qStatus = tStatement.%Prepare(myquery)
    IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  &sql(CALL %SYSTEM_SQL.PTools_exportSQLStats('$IO'))
```

The following example gathers performance statistics on the main module of a query (Action option 2) that was prepared by the current process, then uses the StatsSQLView query to display these statistics:

ObjectScript

```
  DO ##class(%SYSTEM.SQL.PTools).clearSQLStatsALL()
  DO ##class(%SYSTEM.SQL.PTools).setSQLStatsFlagJob(2)
  SET myquery = "SELECT TOP 5 Name,DOB 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}
GetStats
  SET qStatus = tStatement.%Prepare("SELECT * FROM %SYS_PTools.StatsSQLView")
   IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET rsstats = tStatement.%Execute()
  DO rsstats.%Display()
  WRITE !!,"End of SQL Statistics"
```

The following example gathers performance statistics on all modules (Action option 3) of all queries in the USER namespace. When the statistics collection time expires after 1 minute, it resets to Action option 2 and the scope of collecting defaults to 15 (all statistics) on all namespaces:

ObjectScript

```
  DO ##class(%SYSTEM.SQL.PTools).clearSQLStatsALL("USER")
  DO ##class(%SYSTEM.SQL.PTools).setSQLStatsFlagByNS("USER",3,,7,"M:1:2")
  SET myquery = "SELECT TOP 5 Name,DOB 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}
GetStats
  SET qStatus = tStatement.%Prepare("SELECT * FROM %SYS_PTools.StatsSQLView")
   IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET rsstats = tStatement.%Execute()
  DO rsstats.%Display()
  WRITE !!,"End of SQL Statistics",!
TerminateResetStats
  WRITE "returns:  ",##class(%SYSTEM.SQL.PTools).getSQLStatsFlag(),!
  HANG 100
  WRITE "reset to: ",##class(%SYSTEM.SQL.PTools).getSQLStatsFlag()
```

4

Get SQL Performance Help

InterSystems Worldwide Response Center (WRC) offers customer support for analyzing query performance. The WRC can help you finely tune your optimizations to increase the efficiency of your queries. To contact the WRC for support for query analysis, run the Generate Report tool from the Management Portal using either of the following:

- Select to **System Explorer** > **Tools** > **SQL Performance Tools** > **Generate Report**.
- Select to **System Explorer** > **SQL** > **Tools** > **Generate Report**.

4.1 Generate Report

To use the **Generate Report** tool, perform the following steps:

1. You must first get a WRC tracking number from the WRC. You can contact the WRC from the Management Portal by using the **Contact** button found at the top of each Management Portal page. Enter this tracking number in the **WRC Number** area. You can use this tracking number to report the performance of a single query or multiple queries.
2. In the **SQL Statement** area, enter a query text. An **X** icon appears in the top right corner. You can use this icon to clear the **SQL Statement** area. When the query is complete, select the **Save Query** button. The system generates a query plan and gathers runtime statistics on the specified query. Regardless of the system-wide runtime statistics setting, the **Generate Report** tool always collects with Collection Option 3: record statistics for all module levels of the query. Because gathering statistics at this level may take time, it is strongly recommended that you select the **Run Save Query process in the background** check box. This check box is selected by default.

   When a background job is started, the tool displays the message "Please wait...", disables all the fields on the page, and show a new **View Process** button. Clicking the **View Process** button will open the Process Details page in a new tab. From the Process Details page, you can view the process, and may "Suspend", "Resume" or "Terminate" the process. The status of the process is reflected on the Save Query page. When the process is finished, the **Currently Saved Queries** table is refreshed, the **View Process** button disappears, and all the fields on the page are enabled.
3. Perform Step 2 with each desired query. Each query will be added to the **Currently Saved Queries** table. Note that this table can contain queries with the same WRC tracking number, or with different tracking numbers. When finished with all queries, proceed to Step 4.

   For each listed query, you can select the **Details** link. This link opens a separate page that displays the full SQL Statement, the Properties (including the WRC tracking number and the InterSystems IRIS software version), and the Query Plan with performance statistics for each module.

   - To delete individual queries, check the check boxes for those queries from the **Currently Saved Queries** table and then click the **Clear** button.

- To delete all queries associated with a WRC tracking number, select a row from the **Currently Saved Queries** table. The WRC number appears in the **WRC Number** area at the top of the page. If you then click the **Clear** button, all queries for that WRC number are deleted.

4. Use the query check boxes to select the queries you wish to report to the WRC. To select all queries associated with a WRC tracking number, select a row from the **Currently Saved Queries** table, rather than using the check boxes. In either case, you then select the **Generate Report** button. The **Generate Report** tool creates an xml file that includes the query statement, the query plan with runtime statistics, the class definition, and the sql int file associated with each selected query.

   If you select queries associated with a single WRC tracking number, the generated file will have a default name such as `WRC12345.xml`. If you select queries associated with more than one WRC tracking number, the generated file will have the default name `WRCMultiple.xml`.

   A dialog box appears that asks you to specify the location to save the report to. After the report is saved, you can click the **Mail to** link to send the report to WRC customer support. Attach the file using the mail client's attach/insert capability.