InterSystems™
IRIS Data Platform

Using SQL in Productions

Version 2024.1
2024-05-16

Table of Contents

1

Introduction to SQL Adapters

Using an interoperability production to work with data from an external relational data source is possible by executing an SQL query or procedure from within the production. The easiest way to execute the query or procedure is to add a pre-built business service or business operation to your production. These business services and business operations allow your production to communicate with JDBC- or ODBC-compliant databases using built-in SQL adapters. No custom coding is required. However, you also have the option of building a custom business service or business operation that uses these adapters to access the external data source.

Note: When using the SQL adapters, you should be aware of the particular limitations (syntactical or otherwise) of the database to which you are connecting, as well as the database driver that you use to do so.

Note: The SQL adapters used by business services and business operations are clients and perform authentication by passing calls that include the username and password; they do not use authentication from the operating system.

2

Using an SQL Business Service

InterSystems provides two pre-built business services that use SQL to bring data into a production, one for queries and another for stored procedures. To use a business service to run a query, add `EnsLib.SQL.Service.GenericService` to your production. If you want to execute a SQL procedure instead of a query, use `EnsLib.SQL.Service.ProcService`.

These built-in business services use the Inbound SQL Adapter (EnsLib.SQL.InboundAdapter) to access the external data source. In some cases, you might need to build a custom business service that gives you more control over this adapter and how its results are processed.

2.1 Specifying the Data Source Name

To specify the data source that contains the data you want to work with in the production, use the Management Portal to define the following business service settings.

DSN

> This data source name specifies the external data source to which to connect. InterSystems IRIS® distinguishes between these three forms automatically: a defined InterSystems SQL Gateway connection, a JDBC URL, or an ODBC DSN configured in your operating system.
>
> If this name matches the name of a JDBC or ODBC connection configured from the **System Administration** > **Configure** > **Connectivity** > **SQL Gateway Connections** page of the Management Portal, InterSystems IRIS uses the parameters from that specification. If the entry is not the name of a configured connection and it contains a colon (:), it assumes a JDBC URL, otherwise it assumes an ODBC DSN.
>
> The following example shows the name of a DSN that refers to a JDBC URL:
>
> ```
> jdbc:IRIS://localhost:9982/Samples
> ```
>
> The following example shows the name of an ODBC DSN that refers to a Microsoft Access database:
>
> ```
> accessplayground
> ```

If this data source is protected by a password, create production credentials to contain the username and password. Then set the Credentials setting equal to the ID of those credentials; see Specifying Other Runtime Settings for details.

If you are using a JDBC data source, the following settings also apply:

Java Gateway Service

Configuration name of the Java Gateway service controlling the Java Gateway server that this business service uses. The underlying adapter setting is `JGService`.

Important: This setting is *required* for all JDBC data sources, even if you are using a working SQL gateway connection with JDBC. For JDBC connections to work, a business service of type EnsLib.JavaGateway.Service must be present. The business service's SQL adapter requires the name of this configuration item and uses its configured settings to connect to the JVM it manages.

JDBC Driver

JDBC driver class name.

If you use a named SQL Gateway Connection as DSN, this value is optional; but if present, it overrides the value specified in the named JDBC SQL Gateway Connection set of properties.

JDBC Classpath

Classpath for JDBC driver class name, if needed in addition to the ones configured in the Java Gateway Service.

Connection Attributes

An optional set of SQL connection attribute options. For ODBC, they have the form:

attr:*val*,*attr*:*val*

For example, `AutoCommit:1`.

For JDBC, they have the form

attr=*val*;*attr*=*val*

For example, `TransactionIsolationLevel=TRANSACTION_READ_COMMITTED`.

If you use a named JDBC SQL Gateway Connection as DSN, this value is optional; but if present, it overrides the value specified in the named JDBC SQL Gateway Connection set of properties.

2.2 Specifying Inbound Queries

By default, the SQL business service uses the inbound adapter to execute a query periodically. The adapter sends the results, row by row, to the business service.

2.2.1 Specifying the Query

To specify the base query used by the business service, you use the Query setting, which specifies the base query string. It can include the standard SQL ? to represent replaceable parameters, which you specify in a separate setting. Consider the following examples:

```
SELECT * FROM Customer

SELECT p1,p2,p3 FROM Customer WHERE updatetimestamp > ?

SELECT * FROM Sample.Person WHERE ID > ?

SELECT * FROM Sample.Person WHERE Age > ? AND PostalCode = ?
```

2.2.2 Specifying Parameters

The Parameters setting specifies any replaceable parameters in the query string. This setting should equal a comma-separated list of parameter value specifiers, as follows:

```
value,value,value,...
```

For a given value, you can use a constant literal value such as 10 or Gotham City; or you can refer to any of the following:

- You can refer to a property of the adapter. Within the Parameters setting, use the syntax %*property_name*.

- You can refer to a property of the business service. Use the syntax $*property_name*.

  For example, you could add a property named LastTS to the business service class to contain a timestamp. Within the Parameters setting, you would refer to the value of that property as $LastTS

- You can refer to a special persistent value such as &%LastKey, which contains the IDKey value of the last row processed by the adapter.

  Within the Parameters setting, if a parameter name starts with an ampersand (&), InterSystems IRIS assumes that it is a special persistent value.

  Note: For information on initializing these values using a custom business service, see Initializing the Adapter.

2.2.3 Specifying the Maximum Length of String Data

The **VARCHAR LOB Boundary** setting of the business service specifies the maximum length of a string that may be stored in InterSystems IRIS using the **VARCHAR** data type. If a column in the source database contains a string longer than the value you specify, then InterSystems IRIS stores the string using the **LOB** (Large OBject) data type. The corresponding property of the business service's inbound adapter is MaxVarCharLengthAsString.

The default maximum string length is 32767. You can specify a value of -1 to use the maximum string length for InterSystems IRIS, which is currently 3641144 and subject to change in future versions. If you specify a value greater than the maximum string length for InterSystems IRIS, then the current maximum string length for InterSystems IRIS is used.

2.3 SQL Procedure Settings

For additional information about settings that affect how a business service calls a stored procedure, see More About Procedures.

2.4 About Messages

You do not need to create a custom message class to receive the data retrieved from the external data source. By default, the SQL data is placed in a dynamic object with properties for each column of the query. The JSON string describing this object is inserted as the Stream value of an Ens.StreamContainer, allowing it to be sent through the production to other business hosts.

If you decide to develop a custom message class that is used to transport data through the production, the property names must be an exact match with the columns of the SQL query. As a workaround, you can use the SQL As keyword to rename

the column to the property name. Once you have defined your message class, open the Management Portal and use the business service's **Message Class** setting to select it.

For more information about data types and messages, see More About Messages.

3

Using an SQL Business Operation

InterSystems provides two pre-built business operations that use SQL to bring data into a production when they receive an incoming message from another business host. To use a business operation to run a query in response to a message, add `EnsLib.SQL.Operation.GenericOperation` to your production. If you want to execute a SQL procedure instead of a query, use `EnsLib.SQL.Operation.ProcOperation`.

These built-in business operations use the Outbound SQL Adapter (EnsLib.SQL.OutboundAdapter) to access the external data source. In some cases, you might need to build a custom business operation that gives you more control over this adapter and how its results are processed.

By default, a warning is issued the first time that a business operation's query returns multiple rows of results, but this warning is not repeated the next time. You can change that behavior by disabling the business operation's **Only Warn Once** setting. Be aware that the **Only Warn Once** setting does not affect what happens when a property cannot be set (it only generates a warning the first time), but does affect what happens when an incoming stream is truncated to fit into a string property.

3.1 Specifying the Data Source Name

The business operation has a setting that specifies the data source that you want to connect to. When you configure the business operation, you should set an appropriate value for this setting:

DSN

> This data source name specifies the external data source to which to connect. InterSystems IRIS® distinguishes between these three forms automatically: a defined InterSystems SQL Gateway connection, a JDBC URL, or an ODBC DSN configured in your operating system.
>
> If this name matches the name of a JDBC or ODBC connection configured from the **System Administration** > **Configure** > **Connectivity** > **SQL Gateway Connections** page of the Management Portal, InterSystems IRIS uses the parameters from that specification. If the entry is not the name of a configured connection and it contains a colon (:), it assumes a JDBC URL, otherwise it assumes an ODBC DSN.

If this data source is protected by a password, create production credentials to contain the username and password. Then set the Credentials setting equal to the ID of those credentials; see Specifying Other Runtime Settings for details.

If you are using a JDBC data source, the following settings also apply:

JGService

> Configuration name of the Java Gateway service controlling the Java Gateway server this operation uses.

Important: This setting is *required* for all JDBC data sources, even if you are using a working SQL gateway connection with JDBC. For JDBC connections to work, a business service of type EnsLib.JavaGateway.Service must be present. The SQL adapter requires the name of this configuration item and uses its configured settings to connect to the JVM it manages.

JDBCDriver

JDBC driver class name.

If you use a named SQL Gateway Connection as DSN, this value is optional; but if present, it overrides the value specified in the named JDBC SQL Gateway Connection set of properties.

JDBCClasspath

Classpath for JDBC driver class name, if needed in addition to the ones configured in the Java Gateway Service.

ConnectionAttributes

An optional set of SQL connection attribute options. For ODBC, they have the form:

attr:val,attr:val

For example, `AutoCommit:1`.

For JDBC, they have the form

attr=val;attr=val

For example, `TransactionIsolationLevel=TRANSACTION_READ_COMMITTED`.

If you use a named JDBC SQL Gateway Connection as DSN, this value is optional; but if present, it overrides the value specified in the named JDBC SQL Gateway Connection set of properties.

3.2 Specifying a Query

The value for the business operation's **Query** setting is a SQL string (such as `SELECT` or `INSERT INTO`) that is executed when the business operation receives an inbound message from another business host in the production. It can include the standard SQL `?` to represent replaceable parameters.

The **Input Parameters** setting is a comma-separated string specifying values to pass in as the parameters specified in the **Query** setting. You also have the option of passing in a value received as a property of the inbound message. To replace a parameter with the value of a message property, specify the name of the property preceded by the `*` character. You can use this syntax even if the inbound message is a JSON string in an `Ens.StringContainer` or `Ens.StreamContainer`, as long as the specified property is a property of the described dynamic object.

3.3 Specifying Other Runtime Settings

The SQL business operation provides the following additional runtime settings:

Credentials

ID of the production credentials that can authorize a connection to the given DSN. For information on creating production credentials, see *Configuring Productions*.

StayConnected

Specifies whether to keep the connection open between commands, such as issuing an SQL statement or changing an ODBC driver setting.

- If this setting is 0, the SQL outbound adapter disconnects immediately after each command.

- If this setting is positive, it specifies the idle time, in seconds, after the command completes. The adapter disconnects after this idle time.

- If this setting is –1, the adapter auto-connects on startup and then stays connected.

If you are managing database transactions as described in Managing Transactions, do not set StayConnected to 0.

For any settings not listed here, see *Configuring Productions*.

3.4 SQL Procedure Settings

For additional information about settings that affect how a business operation calls a stored procedure, see More About Procedures.

3.5 About Response Messages

Like messages used by a business service, you do not need to develop a custom message class to receive data retrieved by a business operation. By default, a dynamic object is used to return the data to the business host that called the business operation. This dynamic object has properties for each column of the query, and the JSON string describing the object is inserted as the Stream value of an `Ens.StreamContainer`. If the business operation's query returns multiple rows of results, the JSON string contains all of the rows.

If you prefer to use a custom response message rather than the dynamic object, use the business operation's **Response Class** setting to specify the custom message class. The properties of this custom class must match the columns of the SQL query, but you can use the SQL `As` keyword to work around this requirement. The value from a column is placed into the corresponding property of the message. If a Select query returns multiple rows, the message contains values from the first row only.

Regardless of whether you are using a custom message class, when the business operation runs an Update, Insert, or Delete query, the response message contains only one property: `NumRowsAffected`.

For more information about data types and response messages, see More About Messages.

4

More About Messages

For basic information about messages in a production that uses SQL business hosts, see Using a SQL Business Service or Using a SQL Business Operation.

4.1 Incoming Streams

If a production does not define a message class or response class, and the incoming data from a result set column or output parameter is a stream (CLOB or BLOB), the stream is added as the value of the corresponding property in the dynamic object.

Similarly, if the data type of the property of a custom message or response class is a stream, then the incoming stream is simply set into that property. If, however, the data type of the property is a string, then the behavior will depend on the length of the incoming data relative to the MAXLEN of the property, and also on the value of the **Allow Truncating** setting.

If the length of the incoming stream data does not exceed the MAXLEN of the string property, the value is set as a string into the property. If the length of the incoming data does exceed the MAXLEN of the property, then the default behavior is to throw an error and quit. However, if the **Allow Truncating** setting is set to true, then the first MAXLEN characters of the incoming value is set into the property as a string, and a warning is issued. By default, a warning is only issued the first time that the value for any given property needs to be truncated. However, if the **Only Warn Once** setting is disabled, then warnings are issued every time data is truncated. Be aware that the **Only Warn Once** setting does not affect what happens when a property cannot be set (it only generates a warning the first time), but does affect what happens when a business operation's query returns multiple rows of results.

4.2 Incoming Strings

If incoming data from a result set column or output parameter is a string, but the corresponding property of the message or response class is a stream, the value is simply written to the stream.

5
More About Procedures

Both business services and business operations can use a stored procedure to retrieve data from an external data source, and the approach is the similar regardless of whether you are using a service or operation. After adding `EnsLib.SQL.Service.ProcService` or `EnsLib.SQL.Operation.ProcOperation` to your production, use the **Procedure** setting to specify the name of the procedure, including a `?` for each parameter. For example, you could enter `Sample.Stored_Procedure_Test(?,?)`.

Unlike queries, a call to a store procedure can have return values: input parameters can be passed by reference, there can be output parameters, and the entire procedure can return a value. To handle these return values, and differentiate between input and output values, the business service or business operation uses three settings: **Parameters** (business service) or **Input Parameters** (business operation) contain values that are passed to the procedure, **Output Parameter Names** contains property names that are set to values returned in a parameter, and **Input/Output** identifies whether parameters in the procedure call are input parameters, output parameter, or both.

The **Input/Output** setting is key to understanding how the business host handles parameters and return values. It accepts three characters that identify the type of parameter: "`i`" (input), "`o`" (ouput), and "`b`" (both/ByRef). The order of these characters within the **Input/Output** setting corresponds to the order of the parameters of the procedure call. For example, if the procedure call is `Sample.Stored_Procedure_Test(?,?,?)` and the value of **Input/Output** is `ioo`, then the first parameter of the procedure call accepts the value from the **Parameters** setting, and the second and third parameters are values returned by the procedure. These parameter return values are assigned to the names specified in the **Output Parameter Name** setting.

In cases where the entire procedure returns a value, the first character of **Input/Output** should be `o` and the first name in the **Output Parameter Names** setting will be assigned the return value.

As an extended example of how this works, suppose your business operation has the following settings:

Setting	Value
Procedure	`Sample.Stored_Procedure_Test(?,?,?,?)`
Input Parameters	`*Value,x`
Output Parameter Names	`ReturnValue,Response,Stream,TruncatedStream`
Input/Output	`oiboo`

Notice that the value of **Input/Output** is greater than the number of question marks in the procedure because there is also a return value. That return value corresponds to the first string in **Output Parameter Names**. The second value in **Input/Output** is `i` because the first parameter in the procedure – the first `?` – is an input parameter. In this case, we are taking that from the `Value` property of the incoming message. The second parameter is both input and output, meaning that it is passed ByRef. The value `x` is sent in, but then it also corresponds to the second string in **Ouput Parameter Names**, so the response message has its Response property set to the value returned in that parameter. The other two parameters are strictly output

parameters, so nothing is passed into them, and the return values are assigned to the Stream and TruncatedStream properties of the response message.

6

Custom SQL Business Services

This topic describes how to build a custom SQL business service, including a detailed discussion of the SQL inbound adapter (EnsLib.SQL.InboundAdapter) and how to use it in your productions. If you prefer to use a pre-built business service, see Using an SQL Business Service.

6.1 Overall Behavior

First, it is useful to understand the details that you specify for the adapter. The EnsLib.SQL.InboundAdapter class provides runtime settings that you use to specify items like the following:

- A polling interval, which controls how frequently the adapter checks for new input

- The external data source to which the adapter connects

- The ID of the production credentials that provide the username and password for that data source, if needed

- An SQL query to execute

- Optional parameters to use in the query

In general, the inbound SQL adapter (EnsLib.SQL.InboundAdapter) periodically executes a query and then iterates through the rows of the result set, passing one row at a time to the associated business service. The business service, which you create and configure, uses this row and communicates with the rest of the production. More specifically:

1. The adapter regularly executes its **OnTask()** method, which executes the given query. The polling interval is determined by the CallInterval setting.

2. If the query returns any rows, the adapter iterates through the rows in the result set and does the following for each row:

   - If this row has already been processed and has not changed, the adapter ignores it.

     To determine if a given row has already been processed, the adapter uses the information in the KeyFieldName setting; see Processing Only New Rows.

   - If this row has already been processed (as identified by the KeyFieldName setting) and an error occurred, the adapter ignores it until the next restart.

   - Otherwise, the adapter builds an instance of the EnsLib.SQL.Snapshot class and puts the row data into it. This instance is the *snapshot object*. Using the Default Snapshot Object provides details on this object.

     The adapter then calls the internal ProcessInput() method of the associated business service class, passing the snapshot object as input.

3. The internal ProcessInput() method of the business service class executes. This method performs basic production tasks such as setting up a monitor and error logging; these tasks are needed by all business services. You do not customize or override this method, which your business service class inherits.

4. The ProcessInput() method then calls your custom OnProcessInput() method, passing the snapshot object as input. The requirements for this method are described in Implementing the OnProcessInput() Method.

The following figure shows the overall flow:
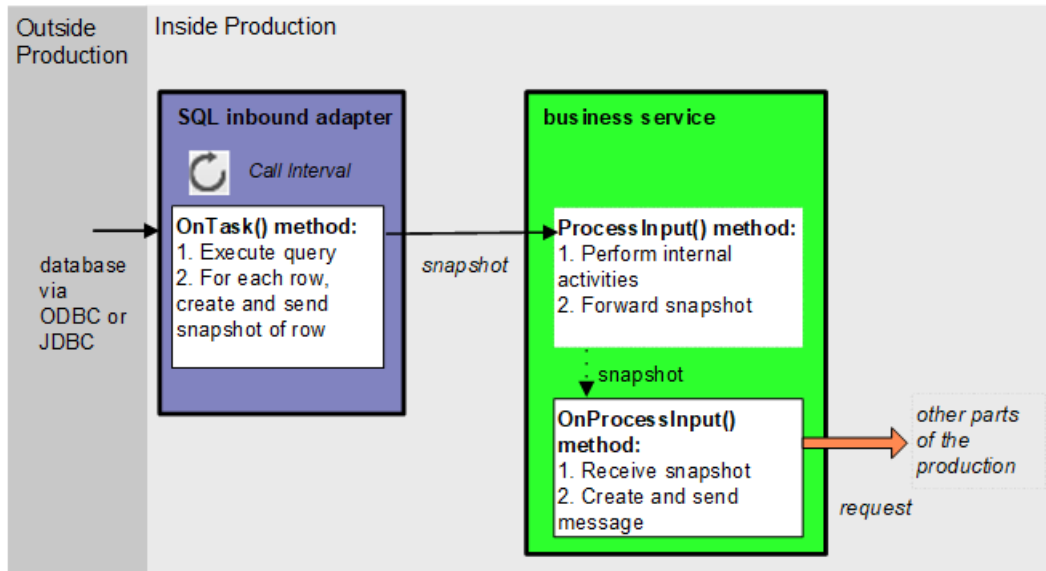
6.2 Creating a Business Service to Use the Adapter

To use this adapter in your production, create a new business service class as described here. Later, add it to your production and configure it. You must also create appropriate message classes, if none yet exist. See Defining Messages.

The following list describes the basic requirements of the business service class:

- Your business service class should extend Ens.BusinessService.

- In your class, the *ADAPTER* parameter should equal EnsLib.SQL.InboundAdapter.

- Your class should implement the OnProcessInput() method, as described in Implementing the OnProcessInput() Method.

- Your class can optionally implement **OnInit()**; see Initializing the Adapter.

- For other options and general information, see Defining a Business Service Class.

The following example shows the general structure that you need:

Class Definition

```
Class ESQL.NewService1 Extends Ens.BusinessService
{

Parameter ADAPTER = "EnsLib.SQL.InboundAdapter";

Method OnProcessInput(pInput As EnsLib.SQL.Snapshot, pOutput As %RegisteredObject) As %Status
{

   Quit $$$ERROR($$$NotImplemented)
}

}
```

6.3 Implementing the OnProcessInput() Method

Within your custom business service class, your **OnProcessInput()** method should have the following signature:

```
Method OnProcessInput(pInput As EnsLib.SQL.Snapshot,
                      pOutput As %RegisteredObject) As %Status
```

Here *pInput* is the *snapshot object* that the adapter sends to this business service; this is an instance of EnsLib.SQL.Snapshot. Also, *pOutput* is the generic output argument required in the method signature.

The **OnProcessInput()** method should do some or all of the following:

1. Create an instance of the request message, which will be the message that your business service sends.

    For information on creating message classes, see Defining Messages.

2. For the request message, set its properties as appropriate, using values in the snapshot object. This object corresponds to a single row returned by your query; for more information, see Using the Default Snapshot Object.

3. Call a suitable method of the business service to send the request to some destination within the production. Specifically, call **SendRequestSync()**, **SendRequestAsync()**, or (less common) **SendDeferredResponse()**. For details, see Sending Request Messages

    Each of these methods returns a status (specifically, an instance of %Status).

4. Optionally check the status of the previous action and act upon it.

5. Optionally examine the response message that your business service has received and act upon it.

6. Return an appropriate status.

The following shows a simple example:

Class Member

```
Method OnProcessInput(pInput As EnsLib.SQL.Snapshot,
   pOutput As %RegisteredObject) As %Status
{
 set req=##class(ESQL.request).%New()
 set req.CustomerID=pInput.Get("CustomerID")
 set req.SSN=pInput.Get("SSN")
 set req.Name=pInput.Get("Name")
 set req.City=pInput.Get("City")
 set sc=..SendRequestSync("ESQL.operation",req,.pOutput)

 quit sc
}
```

Notice that this example uses the **Get()** method of EnsLib.SQL.Snapshot to get data for specific columns; see Using the Default Snapshot Object.

6.4 Using the Default Snapshot Object

When you create a business service class to work with EnsLib.SQL.InboundAdapter, the adapter passes a snapshot object (an instance of EnsLib.SQL.Snapshot) to your custom OnProcessInput() method. This instance contains data for one row of the data returned by your query. Your OnProcessInput() method typically uses the data available in this object.

Note: The EnsLib.SQL.Snapshot class provides properties and methods to manage multiple rows. However, multiple-row snapshots are relevant only for operations that use the outbound adapter.

The following list describes the methods that you are most likely to use within your custom OnProcessInput() method:

Get()

```
method Get(pName As %String, pRow=..%CurrentRow) returns %String
```

Returns the value of the column that has the name *pName*, in the current row (which is the only row in this case).

GetColumnId()

```
method GetColumnId(pName As %String) returns %Integer
```

Returns the ordinal position of the column that has the name *pName*. This method is useful when you work with unfamiliar tables.

GetData()

```
method GetData(pColumn As %Integer, pRow=..%CurrentRow) returns %String
```

Returns the value of the column whose position is specified by *pColumn* in the current row (which is the only row in this case). From left to right, the first column is 1, the second column is 2, and so on.

GetColumnName()

```
method GetColumnName(pColumn As %Integer = 0)
```

Returns the name of the column whose position is specified by *pColumn*.

GetColumnSize()

```
method GetColumnSize(pColumn As %Integer = 0)
```

Returns the size (the width in number of characters) of the database field whose position is specified by *pColumn*.

GetColumnType()

```
method GetColumnType(pColumn As %Integer = 0)
```

Returns the SQL type of the column whose position is specified by *pColumn*, for example, VARCHAR, DATE, or INTEGER.

Note: SQL type names vary between different database vendors.

The following shows how you might use the **Get()** method to extract data from the snapshot and use it to populate the request message:

```
set req=##class(ESQL.request).%New()
set req.CustomerID=pInput.Get("CustomerID")
set req.SSN=pInput.Get("SSN")
set req.Name=pInput.Get("Name")
set req.City=pInput.Get("City")
```

6.5 Initializing the Adapter

To initialize the inbound adapter, customize the **OnInit()** method of your custom business service class. This method is executed during startup of the business host; by default, this method does nothing.

```
Method OnInit() As %Status
```

The most common reason to initialize the adapter is to initialize values for use as parameters of the query, as described in Specifying Inbound Queries. The following subsections list the relevant methods and provide an example.

6.5.1 Initializing Persistent Values

EnsLib.SQL.InboundAdapter provides the following method to initialize persistent values that are saved between restarts of the business service:

```
ClassMethod InitializePersistentValue
            (pConfigName As %String,
             pPersistentValueName As %String = "%LastKey",
             pNewValue As %String)
            As %String
```

Use this to initialize a name-value pair associated with the adapter and then use the name for a parameter of the query. This method checks the current value of the given persistent name-value pair. If the value is currently null, this method sets it equal to *pNewValue*.

By default, if you omit the name, the method initializes the persistent name-value pair `&%LastKey` which contains the IDKey value of the last row processed by the adapter.

In some cases, you might instead need the **InitializeLastKeyValue()**, which initializes the transient adapter property %LastKey. This property is reset each time the business service is started. Also see the class documentation for EnsLib.SQL.InboundAdapter for information on the related methods **SetPersistentValue()** and **GetPersistentValue()**.

6.5.2 Examples

To initialize the `&%LastKey` persistent value, you would customize the **OnInit()** method of your business service to include the following:

Class Member

```
Method OnInit() As %Status
{
   #; initialize persistent last key value
   Do ..Adapter.InitializePersistentValue(..%ConfigName,,0)
   Quit $$$OK
}
```

To initialize the `&TopSales` persistent value, you would customize the **OnInit()** method of your business service to include the following:

Class Member

```
Method OnInit() As %Status
{
    #; must initialize so the query can do a numeric comparison
    Do ..Adapter.InitializePersistentValue(..%ConfigName,"TopSales",0)
    Quit $$$OK
}
```

6.6 Adding and Configuring the Business Service

To add your business service to a production, use the Management Portal to do the following:

1. Add an instance of your custom business service class to the production.

2. Enable the business service.

3. Set the **PoolSize** setting to 1.

    If **PoolSize** is larger than 1, the adapter will process many records twice.

4. Configure the adapter to communicate with a specific external data source. For details about the configuration settings, see Using an SQL Business Service.

5. Run the production.

6.7 Processing Only New Rows

Because it is often undesirable to keep executing a query against the same data, the EnsLib.SQL.InboundAdapter provides several tools that you can use to keep track of the rows that it has processed. This section discusses those tools and then describes several ways to use them in practice.

CAUTION: Be sure to set the **PoolSize** setting equal to 1. If it is larger than 1, the adapter will process many records twice.

6.7.1 Available Tools

The EnsLib.SQL.InboundAdapter provides the following tools to keep track of the rows that it has processed:

- If you specify the KeyFieldName setting, the adapter adds data to an InterSystems IRIS global that indicates which rows it has processed. This setting should refer to a field that contains values that are not reused over time; this field must be in the result set returned by the query. The adapter uses the data in that field to evaluate whether a row has previously been processed.

    Note: If you delete a row, InterSystems IRIS removes its KeyFieldName value from the global that tracks processed rows. If you subsequently add the row back in with the same KeyFieldName value, InterSystems IRIS processes the row again.

- The adapter provides a persistent value, `&%LastKey`, that contains the value of the Key Field Name for the last row that was processed. This special persistent value is saved when you restart the business service.

- The adapter provides a transient property, %LastKey, that contains the value of the KeyFieldName for the last row that was processed. This adapter property is created each time you restart the associated business service.

The latter two options are practical only if KeyFieldName refers to a field that increases monotonically for each new row. Also see Initializing the Adapter.

6.7.2 Practical Ways to Not Reprocess Rows

There are three practical ways that you can ensure that you do not reprocess the same data:

- Use the KeyFieldName setting of the adapter. If specified, this setting should refer to a field that contains values that are not reused over time; this field must be in the result set returned by the query. If you specify a KeyFieldName, the adapter uses the data in that field to evaluate whether a row has previously been processed. Note that if you specify the pseudo field %ID in the SELECT and want to specify this field as the KeyFieldName, you should specify the KeyFieldName as ID and should omit the % (percent sign).

    The default is ID.

    For example, you could specify Query as follows:

    ```
    SELECT ID,Name,Done from Sample.Person
    ```

    And then you could specify KeyFieldName as follows:

    ```
    ID
    ```

    This technique can be inefficient because InterSystems IRIS might select a large number of rows during each polling cycle, and only a small number of those rows might be new.

- Use a query that uses a query parameter that refers to the special persistent value &%LastKey (or the transient adapter property %LastKey). For example, you could specify Query as follows:

    ```
    SELECT ID,Name,Done from Sample.Person WHERE ID>?
    ```

    And then you could specify Parameters as follows:

    ```
    &%LastKey
    ```

    Also see Initializing the Adapter.

- After executing the query, delete the source data or update it so that the query will not return the same rows. To do this, you use the DeleteQuery setting. By default, after the EnsLib.SQL.InboundAdapter executes the main query (the Query setting), it executes the DeleteQuery once for each row returned by the main query.

    This query must include exactly one replaceable parameter (a question mark), which the adapter will replace with value specified by KeyFieldName.

    This query can either delete the source data or can perform an update to ensure that the same rows will not be selected by the main query of the adapter.

    For example, you could specify Query as follows:

    ```
    SELECT ID,Name,Done from Sample.Person WHERE Done=0
    ```

    And then you could specify DeleteQuery as follows:

    ```
    UPDATE Sample.Person SET Done=1 WHERE ID=?
    ```

6.8 Reprocessing Rows

In many cases, it is necessary to notice changes in a row that has previously been processed by the SQL inbound adapter. The easiest way to do this is as follows:

- Include a column in the relevant table to record if a given row has been changed.

- Install an update trigger in the data source to update that column when appropriate.

- Within the query used by the adapter, use the value of this column to determine whether to select a given row.

CAUTION: Be sure to set the **PoolSize** setting equal to 1. If it is larger than 1, the adapter will process many records twice.

6.9 Examples That Use Query Settings

This section provides examples that use the preceding settings.

6.9.1 Example 1: Using KeyFieldName

In the simplest example, we select rows from the `Customer` table. This table has a primary key, the `CustomerID` field, an integer that is automatically incremented for each new record. We are interested only in new rows, so we can use this field as the KeyFieldName. Within our production, we use the following settings for the adapter:

Setting	Value
Query	`SELECT * FROM Customer`
DeleteQuery	*none*
KeyFieldName	`CustomerID`
Parameters	*none*

When the production starts, the adapter will automatically select and process all rows in the `Customer` table. As it processes each row, it creates a trace message if tracing is enabled and adds an entry to the Event Log if logging is enabled; this message will have text like the following:

```
Processing row '216'
```

Here `'216'` refers to the `CustomerID` of the row being processed.

After the production startup, during each polling cycle, the adapter will select all rows but will process only the rows that have a new value in the `CustomerID` field.

6.9.2 Example 2: Using &%LastKey or %LastKey

This example is a variation of the preceding. In this case, the main query selects a subset of the rows, which is more efficient than selecting all the rows.

Setting	Value
Query	`SELECT * FROM Customer WHERE ID>?`
DeleteQuery	*none*
KeyFieldName	`CustomerID`
Parameters	`&%LastKey`

In each polling cycle, the adapter determines the value of `&%LastKey` and passes that value to SQL.

Also see Initializing the Adapter.

Note: When the adapter selects a set of rows, it may or may not process them in the order given by the KeyFieldName. For example, in a given polling cycle, it may select rows with `CustomerID` equal to 101, 102, 103, 104, and 105, but it may process customer 103 last (instead of customer 105). After this polling cycle, the value of `&%LastKey` equals 103. So in the next cycle, the adapter will select customers 104 and 105 again, although it will not reprocess them. (This is still more efficient than reselecting all the rows as in the previous example.) To force the adapter to process the rows in a specific order, include an ORDER BY clause within the query, for example:

```
SELECT * FROM Customer WHERE ID>? ORDER BY CustomerID
```

In this case, the value of `&%LastKey` will always be set to the highest `CustomerID` and no rows will be selected more than once.

6.9.3 Example 3: Using DeleteQuery

In this example, the `Customer` table has a field called `Done`, in which we can record whether the adapter has previously selected a given row.

Setting	Value
Query	`SELECT * FROM Customer WHERE Done=0`
DeleteQuery	`UPDATE Customer SET Done=1 WHERE CustomerID=?`
KeyFieldName	`CustomerID`
Parameters	*none*

In common with the preceding example, this example selects any given row only once.

Tip: Using the delete query can be slow, because this query is executed once for each row that is processed. It is more efficient to perform a batch delete (or batch update) at a regular interval.

6.9.4 Example 4: Working with a Composite Key

In many cases, you might want to treat multiple table fields collectively as the primary key. For example, you might have a table of statistics that includes the fields `Month` and `Year`, and your query might need to treat the month and year together as the unique key for the adapter. In such a case, you would use a query that concatenates the relevant fields and uses the AS clause to provide an alias for the composite field.

For example, with SQL*Server, you could use a query that starts as follows:

```
SELECT Stats, Year||Month as ID ...
```

The result set available in the adapter will have a field named ID, which you can use for KeyFieldName.

Note: The syntax for concatenation depends upon the database with which you are working.

6.9.5 Example 5: No KeyFieldName

In some cases, you might not want to use KeyFieldName. If KeyFieldName is null, the adapter does not distinguish rows and does not skip rows that had either an error or were successfully processed already.

For example:

Setting	Value
Query	`Select * from Cinema.Film Where TicketsSold>?`
DeleteQuery	*none*
KeyFieldName	*none*
Parameters	`&TopSales` (this refers to a special persistent value named `TopSales` that is defined within the **OnProcessInput()** method of the business service)

The business service is as follows:

```
Class Test.SQL.TopSalesService Extends Ens.BusinessService
{

Parameter ADAPTER = "EnsLib.SQL.InboundAdapter";

Parameter REQUESTCLASSES As %String = "EnsLib.SQL.Snapshot";

Method OnInit() As %Status
{
   #; must initialize so the query can do a numeric comparison
   Do ..Adapter.InitializePersistentValue(..%ConfigName,"TopSales",0)
   Quit $$$OK
}

Method OnProcessInput(pInput As EnsLib.SQL.Snapshot,
   Output pOutput As Ens.Response) As %Status
{
   Kill pOutput Set pOutput=$$$NULLOREF

   for j=1:1:pInput.ColCount {
   }
   for i=1:1:pInput.RowCount {
      for j=1:1:pInput.ColCount {
      }
   }
   Set tSales=pInput.Get("TicketsSold")
   Set:tSales>$G($$$EnsStaticAppData(
..%ConfigName,"adapter.sqlparam","TopSales")) ^("TopSales")=tSales
   Quit $$$OK
}

}
```

6.10 Specifying Other Runtime Settings

EnsLib.SQL.InboundAdapter provides the following additional runtime settings.

Call Interval

Specifies the polling interval, in seconds, for the adapter. This specifies how frequently this adapter checks for input.

Upon polling, if the adapter finds input, it creates an appropriate InterSystems IRIS object and passes the object to its associated business service. If several inputs are detected at once, the adapter processes all of them sequentially until no more are found. The adapter sends one request to the business service for each item of input it finds. The adapter then waits for the polling interval to elapse before checking for input again. This cycle continues whenever the production is running and the business service is enabled and scheduled to be active.

It is possible to set a property in the business service so that the adapter delays for the duration of the CallInterval in between processing each input. For details, see Developing Productions.

The default CallInterval is 5 seconds. The minimum is 0.1 seconds.

Credentials

ID of the production credentials that can authorize a connection to the given DSN. For information on creating production credentials, see Configuring Productions.

StayConnected

Specifies whether to keep the connection open between commands, such as issuing an SQL statement or changing an ODBC driver setting.

- If this setting is 0, the adapter will disconnect immediately after each SQL command.

- If this setting is –1, the adapter auto-connects on startup and then stays connected. Use this value, for example, if you are managing database transactions as described in Managing Transactions.

This setting can also be positive (which specifies the idle time after each SQL command, in seconds), but such a value is not useful for the SQL inbound adapter, which works by polling. (If the idle time is longer than the polling interval [CallInterval], the adapter stays connected all the time. If the idle time is shorter than the polling interval, the adapter disconnects and reconnects at every polling interval — meaning that the idle time is essentially ignored.)

For any settings not listed here, see Configuring Productions.

6.11 Resetting Rows Previously Processed by the Inbound Adapter

During development and testing, you might find it useful to reset the adapter for a given business service in order to repeat previous tests. To do so, use one of the following methods described here; these are class methods inherited by EnsLib.SQL.InboundAdapter.

CAUTION: You do not normally use these methods within a live production.

ClearRuntimeAppData()

```
ClassMethod ClearRuntimeAppData(pConfigName As %String)
```

Clears all runtime data for the business service that has the given configured name. Note that you can use the adapter property %ConfigName to access the name of currently configured business service. This data is cleared automatically each time the business service starts.

ClearStaticAppData()

```
ClassMethod ClearStaticAppData(pConfigName As %String)
```

Clears static data for the business service specified by the configured name. This data includes all persistent values associated with the adapter, such as the persistent last key value.

ClearAllAppData()

```
ClassMethod ClearAllAppData(pConfigName As %String)
```

This method just executes the **ClearRuntimeAppData()** and **ClearStaticAppData()** class methods.

7

Custom SQL Business Operations

This topic describes how to build a custom SQL business operation, including a detailed discussion of the SQL outbound adapter (EnsLib.SQL.OutboundAdapter) and how to use it in your productions. If you prefer to use a pre-built business operation, see Using an SQL Business Operation.

7.1 Default Behavior

Within a production, an outbound adapter is associated with a business operation that you create and configure. The business operation receives a message from within the production, looks up the message type, and executes the appropriate method. This method usually executes methods of the associated adapter.

The SQL outbound adapter (EnsLib.SQL.OutboundAdapter) provides settings that you use to specify the data source to connect to and any login details needed for that data source. It also provides methods to perform common SQL activities such as the following:

- Executing queries

- Executing stored procedures

- Performing inserts, updates, and deletes

7.2 Creating a Business Operation to Use the Adapter

To create a business operation to use the EnsLib.SQL.OutBoundAdapter, you create a new business operation class. Later, add it to your production and configure it.

You must also create appropriate message classes, if none yet exist. See Defining Messages.

The following list describes the basic requirements of the business operation class:

- Your business operation class should extend Ens.BusinessOperation.

- The *ADAPTER* parameter should equal EnsLib.SQL.OutboundAdapter.

- The *INVOCATION* parameter should specify the invocation style you want to use, which must be one of the following.

  - **Queue** means the message is created within one background job and placed on a queue, at which time the original job is released. Later, when the message is processed, a different background job will be allocated for the task. This is the most common setting.

- **InProc** means the message will be formulated, sent, and delivered in the same job in which it was created. The job will not be released to the sender's pool until the message is delivered to the target. This is only suitable for special cases.

- Your class should define a *message map* that includes at least one entry. A message map is an XData block entry that has the following structure:

```
XData MessageMap
{
<MapItems>
  <MapItem MessageType="messageclass">
    <Method>methodname</Method>
  </MapItem>
  ...
</MapItems>
}
```

- Your class should define all the methods named in the message map. These methods are known as *message handlers*. In general, these methods will refer to properties and methods of the Adapter property of your business operation.

- For other options and general information, see Defining a Business Operation Class.

The following example shows the general structure that you need:

Class Definition

```
Class ESQL.NewOperation1 Extends Ens.BusinessOperation
{
Parameter ADAPTER = "EnsLib.SQL.OutboundAdapter";

Parameter INVOCATION = "Queue";

Method SampleCall(pRequest As Ens.Request, Output pResponse As Ens.Response) As %Status
{
  Quit $$$ERROR($$$NotImplemented)
}

XData MessageMap
{
<MapItems>
  <MapItem MessageType="Ens.Request">
    <Method>SampleCall</Method>
  </MapItem>
</MapItems>
}
}
```

7.3 Creating Methods to Perform SQL Operations

When you create a business operation class for use with EnsLib.SQL.OutboundAdapter, typically your biggest task is writing message handlers, that is, methods to perform various SQL operations. In general, these methods will refer to properties and methods of the Adapter property of your business operation. For example:

```
 set tSC = ..Adapter.ExecuteUpdate(.numrows,sql)
```

A method might look like the following.

```
/// Insert into NewCustomer table
Method Insert(pReq As ESQL.request, Output pResp As ESQL.response1) As %Status
{
 kill pResp
 set pResp=$$$NULLOREF

 set sql="insert into NewCustomer (Name,SSN,City,SourceID) values (?,?,?,?)"

 //perform the Insert
 set tSC = ..Adapter.ExecuteUpdate
   (.nrows,sql,pReq.Name,pReq.SSN,pReq.City,pReq.CustomerID)

 //create the response message
 set pResp=##class(ESQL.response1).%New()
 set pResp.AffectedRows=nrows

 if 'tSC write " failed ",tSC quit tSC
 quit 1
}
```

To create these methods, you should become familiar with the methods and properties of the EnsLib.SQL.OutboundAdapter class. Creating Adapter Methods for SQL provides more detail about these tools.

7.4 Handling Multiple SQL Statements per Message

The adapter configuration is designed to deal with the simple case where the business operation executes one SQL statement per message it receives. If your business operation needs to execute multiple SQL statements for a given message, use the following style (or similar) in your **OnMessage()** method:

```
OnMessage(..)
{
    Set tStayConn=..Adapter.StayConnected
    Set ..Adapter.StayConnected=-1

    //... your ..Adapter SQL Operations here...

    Set ..Adapter.StayConnected=tStayConn
    If 'tStayConn&&..Adapter.Connected Do ..Adapter.Disconnect()
    Quit tSC
}
```

7.5 Adding and Configuring the Business Operation

To add your business operation to a production, use the Management Portal to do the following:

1. Add an instance of your custom business operation class to the production.

2. Enable the business operation.

3. Configure the adapter to communicate with a specific external data source. For details on these configuration settings, see Using an SQL Business Operation.

4. Run the production.

8

Creating Adapter Methods for SQL

This topic describes how to write adapter methods that perform SQL tasks, by using the tools available within the `EnsLib.SQL` package. Typically you write such methods when you use the outbound adapter.

8.1 Overview and Context

In various cases, you need to write methods that perform SQL tasks. The most common cases are as follows:

- If you use the SQL outbound adapter, you write message handlers and add those methods to the message map of the adapter. Then, for example, if a business operation receives a certain type of message, a message handler could add a record to a specific table.

- If you customize the startup or teardown of a business host, your custom **OnInit()** or **OnTearDown()** methods could initialize or clean out certain tables.

To perform such tasks, your custom methods will use the methods of the SQL inbound and outbound adapters, both of which inherit a core set of methods from the `EnsLib.Common` class. These methods can execute queries, run stored procedures, insert records, and so on.

8.2 Using Parameters

If you use parameters when you run queries, perform updates, or execute procedures, you should obtain information on the ODBC driver you are using. You should look for information on the following:

- Whether this driver supports the ODBC **SQLDescribeParam** function, as most drivers do.

  - If so, you can use the SQL adapter methods **ExecuteQuery()** and **ExecuteUpdate()**. Each of these methods accepts any number of parameter names, calls **SQLDescribeParam**, and uses the resulting information to automatically bind those parameters appropriately.

  - If not, you must use the alternative methods **ExecuteQueryParmArray()** and **ExecuteUpdateParmArray()**. In this case, you must create and pass a multidimensional array that contains the parameters and all their attributes.

- Whether this driver supports the ODBC **SQLDescribeProcedureColumns** function, as most of the major drivers do.

- If so, you can use the SQL adapter method **ExecuteProcedure()**. This method accepts any number of parameter names, calls **SQLDescribeProcedureColumns**, and uses the resulting information to automatically bind those parameters appropriately.

- If not, you must use the alternative method **ExecuteProcedureParmArray()**. In this case, you must create and pass a multidimensional array that contains the parameters and all their attributes.

  If the driver does not support **SQLDescribeProcedureColumns**, you will also need to specify whether each parameter you use is an input, output, or input/output type parameter.

8.2.1 Parameter Attributes

To use parameters in your SQL statements, if the ODBC driver does not support the **SQLDescribeParam** or **SQLDescribeProcedureColumns** function, you will have to create an InterSystems IRIS® multidimensional array that contains the parameters and all their appropriate attributes. InterSystems IRIS uses these values to ask the driver how to bind each parameter appropriately:

- SQL data types — These are generally represented in InterSystems IRIS by integers (SqlType values). The InterSystems IRIS include file EnsSQLTypes.inc contains definitions of the most commonly used values. Here are a few examples:

  - 1 represents SQL_CHAR

  - 4 represents SQL_INTEGER

  - 6 represents SQL_FLOAT

  - 8 represents SQL_DOUBLE

  - 12 represents SQL_VARCHAR

  Note that the include file also lists extended types such as SqlDB2BLOB and SqlDB2CLOB, which are also supported by InterSystems IRIS.

  However, consult the documentation for your database driver to see if it uses any nonstandard values not known to InterSystems IRIS.

- Precision — For a numeric parameter, this generally refers to the maximum number of digits that are used by the data type of the parameter. For example, for a parameter of type CHAR(10), the precision is 10. For a nonnumeric parameter, this generally refers to the maximum length of the parameter.

- Scale — For a numeric parameter, this refers to maximum number of digits to the right of the decimal point. Not applicable to nonnumeric parameters.

8.2.2 Specifying Parameters in an InterSystems IRIS Multidimensional Array

To use the methods **ExecuteQueryParmArray()**, **ExecuteUpdateParmArray()**, and **ExecuteProcedureParmArray()**, you first create an InterSystems IRIS multidimensional array to hold the parameters and their values. Then use this array within the argument list as shown in the method signature. The array can contain any number of parameters, and it must have the following structure:

Node	Contents
arrayname	Must indicate the number of parameters.
arrayname(*integer*)	Value of the parameter whose position is *integer*.
arrayname(*integer*,`"SqlType"`)	SqlType of this parameter, if needed. This is a number that corresponds to an SQL data type. See the preceding section for options. See SqlType and CType Values for further information.
arrayname(*integer*,`"CType"`)	Local InterSystems IRIS type of this parameter, if needed. This is a number that corresponds to an SQL data type. See the preceding section for options. See SqlType and CType Values for further information.
arrayname(*integer*,`"Prec"`)	Precision of this parameter, if needed. See the preceding section. The default is 255.
arrayname(*integer*,`"Scale"`)	Scale of this parameter, if needed. See the preceding section. The default is 0.
arrayname(*integer*,`"IOType"`)	IOType for this parameter, if you need to override the flags in the procedure. This is used only by the **ExecuteProcedureParmArray()** method. • 1 represents an input parameter. • 2 represents an input/output parameter. • 4 represents an output parameter.
arrayname(*integer*,`"SqlTypeName"`)	Used in calls to get parameter values from the driver and in the calculation to compute CType when only the SqlType subscript is given.
arrayname(*integer*,`"LOB"`)	Boolean value that specifies whether this parameter is a large object.
arrayname(*integer*,`"Bin"`)	Boolean value that specifies whether the parameter contains binary data rather than character data.

Important: If you execute multiple queries that use the parameter array, kill and recreate the parameter array before each query.

The methods **ExecuteQueryParmArray()**, **ExecuteUpdateParmArray()**, and **ExecuteProcedureParmArray()** first check to see if the given parameter array has descriptor subscripts. (Specifically InterSystems IRIS checks for the `"CType"` or `"SqlType"` subscript for the first parameter.) Then:

• If the array does not have descriptor subscripts, then the method calls the ODBC function **SQLDescribeParam** or **SQLDescribeProcedureColumns** function as appropriate and uses the values that it returns.

• If the array does have descriptor subscripts, then the method uses them.

Also note that you can prevent **ExecuteProcedure()** from calling **DescribeProcedureColumns** (which it calls by default). To do so, you append an asterisk (`*`) to the end of the *pIO* argument. See Executing Stored Procedures.

8.2.2.1 SqlType and CType Values

You can specify both the `"SqlType"` and `"CType"` subscripts for any parameter. It is simpler to use only the `"SqlType"` subscript.

For any given parameter, the values used are determined as follows:

Scenario	`"SqlType"` subscript	`"CType"` subscript	Actual `"SqlType"` value used	Actual `"CType"` value used
1	Specified	Not specified	Value of `"SqlType"` subscript	Computed automatically from `"SqlType"` value
2	Not specified	Specified	Value of `"SqlType"` subscript (which is automatically defined by copying from the `"CType"` subscript)	
3	Not specified	Not specified	Value determined automatically by querying the data source if possible; otherwise InterSystems IRIS uses 12 (SQL_VARCHAR)	
4	Specified	Specified	Value of `"SqlType"` subscript	Value of `"CType"` subscript

8.3 Executing Queries

You can execute queries within an inbound adapter or within a business service. To execute a query, you use the **ExecuteQuery()** or **ExecuteQueryParmArray()** method of the adapter. These methods use the EnsLib.SQL.GatewayResultSet and EnsLib.SQL.Snapshot helper classes, which differ as follows:

- A result set (an instance of EnsLib.SQL.GatewayResultSet) must be initialized. When it has been initialized, it has a live data connection to a data source.

- In contrast, a snapshot (an instance of EnsLib.SQL.Snapshot) is a static object that you can create and populate in various ways. For example, you can populate it with the data of a result set, either all the rows or a subset of rows (starting at some row position). Using Snapshots discusses other ways to populate a snapshot.

Note: This section discusses how to get result sets and snapshots, rather than how to use them. For information on *using* these objects, see Using Result Sets and Using Snapshots.

8.3.1 Use Modes

When you use the **ExecuteQuery()** or the **ExecuteQueryParmArray()** method, you can receive (by reference) either a result set or a snapshot, depending on how you invoke the method. To use these methods, you do the following:

1. Ensure that the adapter is connected to a DSN.

2. If you want to receive a snapshot object:

   a. Create a new instance of EnsLib.SQL.Snapshot.

   b. Optionally specify values for the FirstRow and MaxRowsToGet properties of that instance.

3. Invoke the **ExecuteQuery()** or the **ExecuteQueryParmArray()** method, passing the following arguments to it:

a. The snapshot instance, if any.

b. A string that contains the query.

c. Parameters as appropriate for the query and for the method (see next section).

If you did not provide a snapshot instance, the method returns a result set. If you did pass a snapshot instance to the method, the method creates a new result set, uses it to populate your snapshot instance (using the values of the FirstRow and MaxRowsToGet properties to choose the rows), and then returns the snapshot.

8.3.2 Syntax for the Methods

To execute a query, use one of the following methods:

ExecuteQuery()

```
Method ExecuteQuery(ByRef pRS As EnsLib.SQL.GatewayResultSet,
                pQueryStatement As %String,
                pParms...) As %Status
```

Executes a query. You provide a query string and any number of parameters. The result is returned by reference in the first argument; the result is an instance of EnsLib.SQL.GatewayResultSet or EnsLib.SQL.Snapshot as described previously.

The second argument is the query statement to execute. This statement can include the standard SQL ? to represent replaceable parameters. Note that the statement should not use UPDATE.

ExecuteQueryParmArray()

```
Method ExecuteQueryParmArray(ByRef pRS As EnsLib.SQL.GatewayResultSet,
                pQueryStatement As %String,
                ByRef pParms) As %Status
```

Executes a query. This method is similar to the preceding method with the exception of how parameters are specified. For this method, you specify the parameters in an InterSystems IRIS multidimensional array (pParms), as described in Specifying Parameters in an InterSystems IRIS Multidimensional Array.

Use the **ExecuteQueryParmArray()** method if you need to specify parameters and if the ODBC driver that you are using does not support the ODBC **SQLDescribeParam** function.

8.3.3 Example

The following shows an example method that executes a query:

```
Method GetPhone(pRequest As ESQL.GetPhoneNumberRequest,
  Output pResponse As ESQL.GetPhoneNumberResponse) As %Status
{
   Set pResponse = ##class(ESQL.GetPhoneNumberResponse).%New()
   //need to pass tResult by reference explicitly in ObjectScript
   //Use an adapter to run a query in the Employee database.
   Set tSC = ..Adapter.ExecuteQuery(.tResult,
"Select "_pRequest.Type_" from Employee where EmployeeID="_pRequest.ID)

 //Get the result
   If tResult.Next() {
     Set pResponse.PhoneNumber = tResult.GetData(1)
   } Else {
   //Handle no phone number for example
     Set pResponse.PhoneNumber = ""
   }
   Quit $$$OK
}
```

8.4 Performing Updates, Inserts, and Deletes

To perform a database update, insert, or delete, use one of the following methods:

ExecuteUpdate()

```
Method ExecuteUpdate(Output pNumRowsAffected As %Integer,
                pUpdateStatement As %String,
                pParms...) As %Status
```

Executes an INSERT, UPDATE, or DELETE statement. You can pass any number of parameters for use in the statement. Notes:

- The number of rows affected is returned as output in the first argument.

- The second argument is the INSERT, UPDATE, or DELETE statement to execute. This statement can include the standard SQL ? to represent replaceable parameters.

ExecuteUpdateParmArray()

```
Method ExecuteUpdateParmArray(Output pNumRowsAffected As %Integer,
                pUpdateStatement As %String,
                ByRef pParms) As %Status
```

Executes an INSERT, UPDATE, or DELETE statement. This method is similar to the preceding method with the exception of how parameters are specified. For this method, you specify the parameters in an InterSystems IRIS multidimensional array (pParms), as described in Specifying Parameters in an InterSystems IRIS Multidimensional Array.

You use the **ExecuteUpdateParmArray()** method if you need to specify parameters and if the ODBC driver that you are using does not support the ODBC **SQLDescribeParam** function.

8.4.1 Example

The following example uses the **ExecuteUpdate()** method:

```
/// Insert into NewCustomer table
Method Insert(pReq As ESQL.request,
    Output pResp As ESQL.response1) As %Status
{
 kill pResp
 set pResp=$$$NULLOREF

 set sql="insert into NewCustomer (Name,SSN,City) values (?,?,?)"

 //perform the Insert
 set tSC = ..Adapter.ExecuteUpdate(.nrows,sql,pReq.Name,pReq.SSN,pReq.City)

 //create the response message
 set pResp=##class(ESQL.response1).%New()
 set pResp.AffectedRows=nrows

 if 'tSC write " failed ",tSC quit tSC
 quit 1
}
```

8.4.2 Example with ExecuteUpdateParmArray

The following example is equivalent to the example for **ExecuteUpdate()**. This one uses the **ExecuteUpdateParmArray()** method:

```
/// Insert into NewCustomer table
Method InsertWithParmArray(pReq As ESQL.request,
   Output pResp As ESQL.response1) As %Status
{
 kill pResp
 set pResp=$$$NULLOREF

 set sql="insert into NewCustomer (Name,SSN,City) values (?,?,?)"

 //set up multidimensional array of parameters
 //for use in preceding query
 set par(1)=pReq.Name
 set par(2)=pReq.SSN
 set par(3)=pReq.City

 //make sure to set top level of array,
 //which should indicate parameter count
 set par=3

 //perform the Insert
 set tSC = ..Adapter.ExecuteUpdateParmArray(.nrows,sql,.par)

 //create the response message
 set pResp=##class(ESQL.response1).%New()
 set pResp.AffectedRows=nrows

 if 'tSC write " failed ",tSC quit tSC
 quit 1
}
```

8.5 Executing Stored Procedures

To execute a stored procedure, use one of the following methods:

ExecuteProcedure()

```
Method ExecuteProcedure(ByRef pResultSnapshots As %ListOfObjects,
           Output pOutputParms As %ListOfDataTypes,
           pQueryStatement As %String,
           pIO As %String = "",
           pInputParms...) As %Status
```

Executes an SQL CALL statement that runs a stored procedure. You can pass any number of parameters. Notes:

- The result is returned by reference in the first argument as a list of EnsLib.SQL.Snapshot objects.

- You can create a list of new instances of EnsLib.SQL.Snapshot and pass the list into the method as the first argument. If you do, the method populates these instances and uses the values of its FirstRow and MaxRowsToGet properties to choose the set of rows that each will represent. The method then returns the list of instances.

- The second argument is the list of the output values of all scalar output and input/output parameters. If the procedure returns a scalar return value and your statement retrieves it, this value will be the first output value.

- The third argument is the SQL CALL statement that runs a stored procedure. This statement can include the standard SQL ? to represent replaceable parameters.

    Important: The name of the stored procedure is case-sensitive. Also, make sure that the *pQueryStatement* statement supplies an argument for every input or input/output parameter that the SQL query requires.

- The fourth (optional) argument indicates the type (input, output, or input/output) for each parameter. If you specify this argument, use a string that consists of the characters i, o, and b; the character at a given position indicates the type of the corresponding parameter. For example, iob means that the first parameter is input, the second parameter is output, and the third parameter is both input and output.

> **Tip:** By default, the adapter calls the ODBC function **DescribeProcedureColumns** to get information about the parameters and logs warnings if the parameter types specified here are different from the types returned by that function. To prevent the adapter from making this check, append an asterisk (*) to the end of this string.

Not all database support all these types of parameters. Be sure to use only the types that are supported by the database to which you are connecting.

ExecuteProcedureParmArray()

```
Method ExecuteProcedureParmArray(ByRef pResultSnapshots As %ListOfObjects,
                Output pOutputParms As %ListOfDataTypes,
                pQueryStatement As %String,
                pIO As %String = "",
                ByRef pIOParms) As %Status
```

Executes an SQL CALL statement that runs a stored procedure. This method is similar to the preceding method with the exception of how parameters are specified. For this method, you specify the parameters in an InterSystems IRIS multidimensional array (*pParms*), as described in Specifying Parameters in an InterSystems IRIS Multidimensional Array.

You use the **ExecuteProcedureParmArray()** method if you need to specify parameters and if the ODBC driver that you are using does not support the ODBC **SQLDescribeParam** function.

Also note:

- For a given parameter, if you specify the input/output type within the *pIOParms* array *and* within the *pIO* argument, the type given in *pIOParms* array takes precedence.

- If you specify any input/output types within the *pIOParms* array, then for all output parameters, be sure to leave the corresponding array nodes undefined.

If you have configured your SQL adapter to use JDBC through the Java Gateway, output parameters that have a large object value are returned as streams. For compatibility with older versions of InterSystems IRIS, you can set a global to return these large object output parameters as strings. But even with this global set, if the object exceeds the size allowed for a string, it is returned as a stream. To set this compatibility behavior for a configuration item SQLservice, set the global ^Ens.Config("JDBC","LOBasString","SQLservice") to 1.

8.5.1 Example

The following code executes a stored procedure that has three parameters: an output parameter, an input parameter, and another output parameter. The input parameter is extracted from the Parameters property of the request message: pReq.Parameters.GetAt(1). The output parameters are ignored.

```
Set tQuery="{ ?=call Sample.Employee_StoredProcTest(?,?) }"
Set tSC = ..Adapter.ExecuteProcedure(.tRTs,.tOutParms,tQuery,"oio",pReq.Parameters.GetAt(1))
Set tRes.ParametersOut = tOutParms
```

In this example, tRTs represents a result set that was previously created.

8.6 Specifying Statement Attributes

When using the SQL adapters, you can specify any driver-dependent statement attributes. To do so:

- If the connection has not been established, set the StatementAttrs property of the adapter equal to a comma-separated list of attribute-value pairs as follows:

```
attribute:value,attribute:value,attribute:value,...
```

  All subsequently created statements will inherit these attributes.

  For example:

```
Set ..Adapter.StatementAttrs = "QueryTimeout:10"
```

- If the connection has already been established, call the **SetConnectAttr()** method of the adapter. This method takes two arguments (the attribute name and the desired value) and returns a status. For example:

```
Set tout= ..Adapter.SetConnectAttr("querytimeout",10)
```

  If a network error is detected, by default, the adapter tries to reconnect and start over. If you are setting connection attributes such as AutoCommit, do the following so that this reconnect/retry logic can occur: test the status returned from **SetConnectAttr()** and return that status value from the business operation in the case of an error.

Note: It is your responsibility to ensure that the attributes you use are supported by the ODBC driver for the database to which you are connecting. It is beyond the scope of the InterSystems documentation to attempt to compile any such list.

The most useful places to set statement attributes are as follows:

- Within a message handler method of a business operation, if you use the SQL outbound adapter.

- Within the **OnInit()** method of a business host.

8.7 Managing Transactions

The SQL adapters provide the following methods that you can use to manage formal database transactions:

SetAutoCommit()

```
Method SetAutoCommit(pAutoCommit) As %Status [ CodeMode = expression ]
```

Sets autocommit on or off for this adapter connection. This works only after the DSN connection is established.

If you want to set this at connect time, customize the **OnInit()** method of your business service or operation. In your custom method, set the ConnectAttrs property.

If you switch on autocommit, do not set StayConnected to 0. This setting specifies whether to stay connected to the remote system between handling commands:

- For details on how the SQL inbound adapter uses this setting, see Specifying Other Runtime Settings for the SQL Inbound Adapter.

- For details on how the SQL outbound adapter uses this setting, see Specifying Other Runtime Settings for the SQL Outbound Adapter.

If a network error is detected, by default, the adapter tries to reconnect and start over. If you are setting connection attributes such as AutoCommit, do the following so that this reconnect/retry logic can occur: test the status returned from **SetAutoCommit()** and return that status value from the business operation in the case of an error.

Commit()

```
Method Commit() As %Status
```

Commits all database activities (within this adapter process) since the last commit.

Rollback()

```
Method Rollback() As %Status
```

Rolls back all database activities (within this adapter process) since the last commit.

The following example shows a simple transaction that uses the preceding methods. Of course, production-quality code includes robust error handling. For example, you could wrap these methods in the `Try` block of a Try-Catch construct, then place the `Rollback` method in the `Catch` block to roll back the transaction in the event of an error.

Class Member

```
Method TransactionExample(pRequest As common.examples.msgRequest2,
  Output pResponse As common.examples.msgResponse) As %Status
{
  #include %occStatus
  //initialize variables and objects
  set tSC = $$$OK
  set pResponse = ##class(common.examples.msgResponse).%New()

  #; start the transaction. Set autocommit to 0
  set tSC = ..Adapter.SetAutoCommit(0)

  //Example UPDATE, INSERT, DELETE
  set tQueryIns="insert into common_examples.mytable(name,age,datetime)"
              _" values ('SAMPLE"_$random(9999)_"',40,'"_$zdt($h,3)_"')"

  set tSC = ..Adapter.ExecuteUpdate(.tAffectedRows,tQueryIns)

  // finalize transaction
  set tSC=..Adapter.Commit()

  return $$$OK
}
```

Note: It is important to consider the database activities that make up a given transaction. If these activities are contained within a single business host, you can just use the preceding methods to set up transaction management. However, if the database activities are contained in multiple business hosts, you must write code (typically within a business process) to simulate a true rollback.

8.8 Managing the Database Connection

To manage the database connection of an adapter, you can use the following properties and methods of the adapter.

8.8.1 Properties

The following properties control or provide information about the database connection:

Connected

> %Boolean

> This read-only property indicates if the adapter is currently connected.

ConnectAttrs

> %String

> An optional set of SQL connection attribute options. For ODBC, they have the form:

> *attr:val,attr:val*

> For example, `AutoCommit:1`.

> For JDBC, they have the form

> *attr=val;attr=val*

> For example, `TransactionIsolationLevel=TRANSACTION_READ_COMMITTED`.

> Set this property in the **OnInit()** method of your business operation or business service to specify the options to use at connection time.

ConnectTimeout

> %Numeric

> This property specifies the number of seconds to wait on each connection attempt. The default value is 5.

StayConnected

> %Numeric

> This property specifies whether to stay connected to the remote system:

> - For information on how the SQL inbound adapter uses this setting, see Specifying Other Runtime Settings for the SQL Inbound Adapter.

> - For information on how the SQL outbound adapter uses this setting, see Specifying Other Runtime Settings for the SQL Outbound Adapter.

DSN

> %String

> This data source name specifies the external data source to connect to. The following example shows the name of a DSN that refers to a Microsoft Access database:

>  ```
>  accessplayground
>  ```

8.8.2 Methods

Use the following methods to manage the database connection:

Connect()

>  ```
>  Method Connect(pTimeout As %Numeric = 30) As %Status
>  ```

> Connects to the data source given by the current value of the DSN property.

Disconnect()

```
Method Disconnect() As %Status
```

Disconnects from the data source.

TestConnection()

```
Method TestConnection()
```

Tests the connection to the data source.

The adapter classes also provide several setter methods that you can use to set the properties listed in the preceding section.

9

Using Result Sets (SQL Adapters)

The EnsLib.SQL.GatewayResultSet class represents a special-purpose result set for use by an SQL adapter in a production. An initialized instance of this class has a live data connection to a data source. The class provides methods to examine the contents of the result set as well as a method to return a static snapshot.

This topic describes how to use the EnsLib.SQL.GatewayResultSet class.

Note that you can also get a snapshot that contains rows from the result set; see Using Snapshots.

9.1 Creating and Initializing a Result Set

To create and initialize an SQL result set:

1. Within an SQL adapter (either EnsLib.SQL.InboundAdapter or EnsLib.SQL.OutboundAdapter), connect to a DSN.

2. Use the **ExecuteQuery()** or the **ExecuteQueryParmArray()** method of the adapter. You will receive, by reference, an instance of EnsLib.SQL.GatewayResultSet.

Note: If you just use the **%New()** class method, you can create a result set, but it will not be initialized and cannot contain any data. To initialize the result set, use the procedure described here.

9.2 Getting Basic Information about the Result Set

The following properties of EnsLib.SQL.GatewayResultSet provide basic information about a result set:

- The ColCount property indicates the number of columns in the result set.

- The QueryStatement property indicates the query statement used by this result set.

9.3 Navigating the Result Set

A result set consists of rows of data. You can use the following methods to navigate through the rows:

Next()

```
method Next(ByRef pSC As %Status) returns %Integer
```

Advances the cursor to the next row and caches the row data. Returns 0 if the cursor is at the end of the result set.

SkipNext()

```
method SkipNext(ByRef pSC As %Status) returns %Integer
```

Advances the cursor to the next row. Returns 0 if the cursor is at the end of the result set.

9.4 Examining the Current Row of the Result Set

Use the following methods to examine the current row of the result set:

Get()

```
method Get(pName As %String) returns %String
```

Returns the value of the column that has the name *pName*, in the current row.

GetData()

```
method GetData(pColumn As %Integer) returns %String
```

Returns the value of the column whose position is specified by *pColumn* in the current row.

GetColumnName()

```
method GetColumnName(pColumn As %Integer = 0)
```

Returns the name of the column whose position is specified by *pColumn*.

Note: If the source data contains any unnamed columns, the result set automatically provides names for these columns in the following form: xCol_*n*

10

Using Snapshots (SQL Adapters)

The EnsLib.SQL.Snapshot class represents a static object that you create and populate in various ways. This object is meant for use by an SQL adapter in a production. The class provides methods for examining the data; more methods are available for this object than for the result set. This topic describes how to use the EnsLib.SQL.Snapshot class.

10.1 Creating a Snapshot

When you use the SQL inbound adapter, by default, you automatically receive snapshot objects within your business service. For each row in your query, the adapter creates a snapshot object and sends it as an argument when it calls the **ProcessInput()** method of the business service. As noted previously, by default, this snapshot contains only a single row.

10.1.1 Creating a Snapshot from a Live Connection

In most cases, you will probably have a live connection to the data source. Specifically, you start with an SQL adapter (either EnsLib.SQL.InboundAdapter or EnsLib.SQL.OutboundAdapter). Within the adapter, connect to a DSN. Then you can do any of the following:

- Use the **ExecuteProcedure()** or **ExecuteProcedureParmArray()** method of the adapter. Each of these returns the results as a snapshot.

  These methods are discussed in Using the Result Sets.

- Create a result set (see Using the Result Sets) and then use the **GetSnapshot()** method of the result set. This method has the following signature.

  ```
  method GetSnapshot(ByRef pSnap As EnsLib.SQL.Snapshot,
      pFetchAll As %Boolean = 0) returns %Status
  ```

  Returns a snapshot object by reference in the first argument. If you pass an existing snapshot object to the method, the method uses the FirstRow and MaxRowsToGet properties of that object to determine which rows to place in the snapshot. Otherwise, the method uses the default values.

10.1.2 Creating a Snapshot from Static Data

You can also create a snapshot from static data, without having a connection to a DSN. To do so, use any of the following techniques:

- Use the **CreateFromFile()**, **CreateFromStream()**, or **CreateFromResultSet** class method.

- Create a new instance of a snapshot (via the **%New()** class method), and then use the **ImportFile()**, **ImportFromStream()**, or **ImportFromResultSet()** method.

The following list provides the details for these methods, all of which are in the EnsLib.SQL.Shapshot class:

CreateFromFile()

```
classmethod CreateFromFile(pFilename As %String,
    pRowSeparator As %String,
    pColumnSeparator As %String,
    pColumnWidths As %String,
    pLineComment As %String,
    pStripPadChars As %String,
    pColNamesRow As %Integer,
    pFirstRow As %Integer,
    pMaxRowsToGet As %Integer,
    Output pStatus As %Status) as Snapshot
```

Creates a new snapshot object and loads it with data from a table-formatted text file. The arguments are as follows:

- *pFilename* specifies the name of the file to import. This is the only required argument.

- *pRowSeparator* is one of the following:

    – The character that separates one row from the next row. The default is a line feed character.

    – A number, preceded by a minus sign, that indicates the line length in characters.

- *pColumnSeparator* is one of the following:

    – The character that separates one column from the next column. There is no default character.

    – The number 0, which means that the columns are determined by the *pColumnWidths* argument; see the next argument.

    – A number, preceded by a minus sign, that indicates the number of initial characters to skip in each row. In this case, the columns are determined by the *pColumnWidths* argument; see the next argument.

- *pColumnWidths* is one of the following:

    – A comma-separated list of column widths (number of characters), if the fields in the file are positional.

    – The number of columns, if the file uses column separators.

- *pLineComment* specifies a string after which the rest of a row should be ignored. Within a given row, after this string is found, the snapshot does not parse the rest of the row into columns.

- *pStripPadChars* means characters to strip from the beginning and end of a field. The default is the space character.

- *pColNamesRow* specifies the index of the row that contains column names, if any.

- *pFirstRow* specifies the index of the first row (from the file) to include in the snapshot.

- *pMaxRowsToGet* specifies the maximum number of rows to include in the snapshot.

- *pStatus* is the status that the method returns when it attempts to create the snapshot.

CreateFromStream()

```
classmethod CreateFromStream(pIOStream As %IO.I.CharacterStream,
    pRowSeparator As %String,
    pColumnSeparator As %String,
    pColumnWidths As %String,
    pLineComment As %String,
    pStripPadChars As %String,
    pColNamesRow As %Integer,
    pFirstRow As %Integer,
    pMaxRowsToGet As %Integer,
    Output pStatus As %Status) as Snapshot
```

Creates a new snapshot object and loads it with data from a table-formatted stream. See the comments for **CreateFromFile()**.

CreateFromResultSet

```
classmethod CreateFromResultSet(pRS,
  pLegacyMode As %Integer = 1,
  pODBCColumnType As %Boolean = 0,
  pFirstRow As %Integer,
  pMaxRowsToGet As %Integer,
  Output pStatus As %Status) as Snapshot
```

Creates a new snapshot object and loads it with data from a result set. See the comments for **CreateFromFile()** and for **ImportFromResultSet**.

ImportFile()

```
 method ImportFile(pFilename As %String,
  pRowSeparator As %String = $C(10),
  pColumnSeparator As %String = $C(9),
  pColumnWidths As %String = "",
  pLineComment As %String = "",
  pStripPadChars As %String = " "_$C(9),
  pColNamesRow As %Integer = 0) as %Status
```

Imports data from a table-formatted text file. See the comments for **CreateFromFile()**.

ImportFromStream()

```
method ImportFromStream(pIOStream As %IO.I.CharacterStream,
    pRowSeparator As %String = $C(10),
    pColumnSeparator As %String = $C(9),
    pColumnWidths As %String = "",
    pLineComment As %String = "",
    pStripPadChars As %String = " "_$C(9),
    pColNamesRow As %Integer = 0) as %Status
```

Here *pIOStream* is the stream to import. See the comments for **CreateFromFile()**.

ImportFromResultSet()

```
method ImportFromResultSet(pRS,
    pLegacyMode As %Integer = 1,
    pODBCColumnType As %Boolean = 0) as %Status
```

Imports a result set into a snapshot instance. The arguments are as follows:

- *pRS* is an instance of EnsLib.SQL.GatewayResultSet, or a result set in the %SQL package such as %SQL.StatementResult or %SQL.ISelectResult (%SQL.IResult).

- *pLegacyMode* specifies how to search for meta data. If this argument is 0, then InterSystems IRIS® first tries to use %GetMetadata. This leads to different source of metadata for legacy result set classes. The default is 1, which maintains previous behavior while still supporting %SQL.* and older classes.

- *pODBCColumnType* controls how the ColumnType is set. If *pODBCColumnType* is 1, then ColumnType text is set to the ODBC type column type text and not the clientType.

10.1.2.1 Example

Consider a file that has the following contents:

```
col1,col2,col3
value A1,value A2,value A3
value B1,        value B2          ,value B3
```

The following code reads this file, uses it to create a snapshot, and writes simple comments to the Terminal. Notice that the only arguments used are the filename and the column separator:

ObjectScript

```
set filename="c:/demo.txt"
set snap=##class(EnsLib.SQL.Snapshot).%New()
do snap.ImportFile(filename,,",")
d show
quit


show
 w "number of rows in snapshot=",snap.RowCount,!
 while snap.Next()
  {
      w "current row=",snap.%CurrentRow,!
      w "data in first column=",snap.GetData(1),!
      w "data in second column=",snap.GetData(2),!
      w "data in third column=",snap.GetData(3),!
  }
 quit
```

The output from this routine is as follows:

```
number of rows in snapshot=3
current row=1
data in first column=col1
data in second column=col2
data in third column=col3
current row=2
data in first column=value A1
data in second column=value A2
data in third column=value A3
current row=3
data in first column=value B1
data in second column=value B2
data in third column=value B3
```

Notice that line feeds are used by default as row separators. Also notice that by default, leading and trailing spaces are removed from each field.

10.1.3 Creating a Snapshot Manually

You can also create a snapshot manually, as follows:

1. Create a new instance of a snapshot (via the **%New()** class method).

2. Use the **SetColNames()**, **SetColSizes()**, and **SetColTypes()** methods to specify the names, sizes, and types of the columns.

3. Use the **AddRow()** method to add a row of data.

The following list provides the details for these methods, all of which are in the EnsLib.SQL.Shapshot class:

AddRow()

```
method AddRow(pCol...) returns %Status
```

Adds a row that contains the given data. The argument list is the row data, field by field. For example, the following adds a row to a snapshot. In this case, the column names are ID, Name, and DOB, respectively:

```
set sc=snapshot.SetColNames("1023","Smith,Bob","06-06-1986")
```

SetColNames()

```
method SetColNames(pColName...) returns %Status
```

Sets the names of the columns, in the order given by the arguments. For example, the following sets the column names as ID, Name, and DOB, respectively:

```
set sc=snapshot.SetColNames("ID","Name","DOB")
```

SetColSizes()

```
method SetColSizes(pColSize...) returns %Status
```

Sets the sizes of the columns (the width in number of characters), in the order given by the arguments.

SetColTypes()

```
method SetColTypes(pColType...) returns %Status
```

Sets the types of the columns, in the order given by the arguments.

Note: Remember that the SQL type names vary between different database vendors. Use the type names that are appropriate for the database with which you are working. The **SetColTypes()** method does not perform any checking of your type names.

10.2 Getting Basic Information about the Snapshot

The following properties of the snapshot provide basic information:

- The %CurrentRow property is an integer that indicates the current row.

- The AtEnd property is true if the current row is the last row; otherwise it is false.

- The ColCount properties indicates the number of columns in the snapshot.

- The RowCount properties indicates the number of columns in the snapshot. This property counts only the rows that do not start with the comment string, if any. To create a snapshot that includes comments, use the **CreateFromFile()** and related methods, and specify a value for the *pLineComment* argument. A row is counted if it begins without the comment string but includes the comment string in a later position.

10.3 Navigating the Snapshot

A snapshot consists of rows of data. You can use the following methods to navigate through the rows:

Next()

```
method Next(ByRef pSC As %Status) returns %Integer
```

Advances the cursor to the next row. Returns 0 if the cursor is at the end of the snapshot.

Rewind()

```
method Rewind() returns %Status
```

Returns the cursor to the first row of the snapshot.

10.4 Examining the Current Row of the Snapshot

Use the following methods to examine the current row of the snapshot:

Get()

```
method Get(pName As %String, pRow=..%CurrentRow) returns %String
```

Returns the value of the column that has the name *pName*, in the indicated row (by default, the current row).

GetData()

```
method GetData(pColumn As %Integer, pRow=..%CurrentRow) returns %String
```

Returns the value of the column whose position is specified by *pColumn* in the indicated row (by default, the current row).

GetColumnName()

```
method GetColumnName(pColumn As %Integer = 0)
```

Returns the name of the column whose position is specified by *pColumn*.

GetColumnId()

```
method GetColumnId(pName As %String) returns %Integer
```

Returns the ordinal position of the column that has the name *pName*. This method is useful when you work with unfamiliar tables.

GetColumnSize()

```
method GetColumnSize(pColumn As %Integer = 0)
```

Returns the size (the width in number of characters) of the database field whose position is specified by *pColumn*.

GetColumnType()

```
method GetColumnType(pColumn As %Integer = 0)
```

Returns the type of the column whose position is specified by *pColumn*.

Note: SQL type names vary between different database vendors.

10.5 Resetting a Snapshot

If you have an existing snapshot object, you can clear the data and definitions from it. To do so, use the **Clean()** method, which returns a status. This is slightly more efficient than destroying the snapshot and creating a new one via **%New()**.