Article
Eduard Lebedyuk · Feb 5, 2016 11m read

Class Queries in InterSystems IRIS

Class Queries in InterSystems IRIS (and Cache, Ensemble, HealthShare) is a useful tool that separates SQL queries from Object Script code. Basically, it works like this: suppose that you want to use the same SQL query with different arguments in several different places.In this case you can avoid code duplication by declaring the query body as a class query and then calling this query by name. This approach is also convenient for custom queries, in which the task of obtaining the next row is defined by a developer. Sounds interesting? Then read on!

Basic class queries

Simply put, basic class queries allow you to represent SQL SELECT queries. SQL optimizer and compiler handle them just as they would standard SQL queries, but they are more convenient when it comes to executing them from Caché Object Script context. They are declared as Query items in class definitions (similar to Method or Property) in the following way:

- Type: %SQLQuery
- All arguments of your SQL query must be listed in the list of arguments
- Query type: SELECT
- Use the colon to access each argument (similar to static SQL)
- Define the ROWSPEC parameter which contains information about names and data types of the output results along with the order of fields
- (Optional) Define the CONTAINID parameter which corresponds to the numeric order if the field containing ID. If you don't need to return ID, don't assign any values to CONTAINID
- (Optional) Define the COMPILEMODE parameter which corresponds to the similar parameter in static SQL and specifies when the SQL expression must be compiled. When this parameter is set to IMMEDIATE (by default), the query will be compiled simultaneously with the class. When this parameter is set to DYNAMIC, the query will be compiled before its first execution (similar to dynamic SQL)
- (Optional) Define the SELECTMODE parameter which specifies the format of the query results
- Add the SqlProc property, if you want to call this query as an SQL procedure.
- Set the SqlName property, if you want to rename the query. The default name of a query in SQL context is as follows: PackageName.ClassNameQueryName
- Caché Studio provides the built-in wizard for creating class queries

Sample definition of the Sample.Person class with the ByName query which returns all user names that begin with a specified letter

```
Class Sample.Person Extends %Persistent
{
Property Name As %String;
Property DOB As %Date;
Property SSN As %String;
Query ByName(name As %String = "") As %SQLQuery
    (ROWSPEC="ID:%Integer,Name:%String,DOB:%Date,SSN:%String",
     CONTAINID = 1, SELECTMODE = "RUNTIME",
     COMPILEMODE = "IMMEDIATE") [ SqlName = SP_Sample_By_Name, SqlProc ]
{
SELECT ID, Name, DOB, SSN
FROM Sample.Person
```

```
WHERE (Name %STARTSWITH :name)
ORDER BY Name
}
}
```

You can call this query from Caché Object Script in the following way:

```
Set statement=##class(%SQL.Statement).%New()
Set status=statement.%PrepareClassQuery("Sample.Person","ByName")
If $$$ISERR(status) {
    Do $system.OBJ.DisplayError(status)
}
Set resultset=statement.%Execute("A")
While resultset.%Next() {
    Write !, resultset.%Get("Name")
}
```

Alternatively, you can obtain a resultset using the automatically generated method queryNameFunc:

```
Set resultset = ##class(Sample.Person).ByNameFunc("A")
While resultset.%Next() {
    Write !, resultset.%Get("Name")
}
```

This query can also be called from SQLcontext in these two ways:

```
Call Sample.SP_Sample_By_Name('A')
Select * from Sample.SP_Sample_By_Name('A')
```

This class can be found in the SAMPLES default Caché namespace And that's all about simple queries. Now let's proceed to custom ones.

Custom class queries

Though basic class queries work fine in most cases, sometimes it is necessary to execute full control over the query behavior in applications, e.g.:

- Sophisticated selection criteria. Since in custom queries you implement a Caché Object Script method that returns the next row on your own, these criteria can be as sophisticated as you require.
- If data is accessible only via API in a format that you don't want to use
- If data is stored in globals (without classes)
- If you need to escalate rights in order to access data
- If you need to call an external API in order to access data
- If you need to gain access to the file system in order to access data
- You need to perform additional operations before running the query (e.g. establish a connection, check permissions, etc.)

So, how do you create custom class queries? First of all, you should define 4 methods which implement the entire workflow for your query, from initialization to destruction:

- queryName — provides information about a query (similar to basic class queries)
- queryName**Execute** — constructs a query

- queryName**Fetch** — obtains the next row result of a query
- queryName**Close** — destructs a query

Now let's analyze these methods in more detail.

The queryName method

The queryName method represents information about a query.

- Type: %Query
- Leave body blank
- Define the ROWSPEC parameter which contains the information about names and data types of the output results along with the field order
- (Optional) Define the CONTAINID parameter which corresponds to the numeric order if the field containing ID. If you don't return ID, don't assign any value to CONTAINID

For example, let's create the AllRecords query (queryName = AllRecords, and the method is simply called AllRecords) which will be outputting all instances of the new persistent class Utils.CustomQuery, one by one. First, let's create a new persistent class Utils.CustomQuery:

```
Class Utils.CustomQuery Extends (%Persistent, %Populate){
Property Prop1 As %String;
Property Prop2 As %Integer;
}
```

Now let's write the AllRecords query:

```
Query AllRecords() As %Query(CONTAINID = 1, ROWSPEC = "Id:%String,Prop1:%String,Prop2
:%Integer") [ SqlName = AllRecords, SqlProc ]
{
}
```

The queryNameExecute method

The queryNameExecute method fully initializes a query. The signature of this method is as follows:

```
ClassMethod queryNameExecute(ByRef qHandle As %Binary, args) As %Status
```

where:

- qHandle is used for communication with other methods of the query implementation
- This method must set qHandle into the state which will then be passed to the queryNameFetch method
- qHandle can be set to OREF, variable or a multi-dimensional variable
- args are additional parameters passed to the query. You can add as many args as you need (or don't use them at all)
- The method must return query initialization status

But let's get back to our example. You are free to iterate through the extent in multiple ways (I will describe the basic working approaches for custom queries below), but as for this example let's iterate through the global using the $Order function. In this case, qHandle will be storing the current ID, and since we don't need any additional arguments, the arg argument is not required. The result looks like this:

```
ClassMethod AllRecordsExecute(ByRef qHandle As %Binary) As %Status {
    Set qHandle = ""    Quit $$$OK
```

```
}
```

The queryNameFetch method

The queryNameFetch method returns a single result in $List form. The signature of this method is as follows:

```
ClassMethod queryNameFetch(ByRef qHandle As %Binary, ByRef Row As %List, ByRef AtEnd
As %Integer = 0) As %Status [ PlaceAfter = queryNameExecute ]
```

where:

- qHandle is used for communication with other methods of the query implementation
- When the query is executed, qHandle is being assigned values specified by queryNameExecute or by previous call of queryNameFetch.
- Row will be set either to a value of %List or to an empty string, if all data has been processed
- AtEnd must be set to 1, once the end of data is reached.
- The PlaceAfter keyword identifies the method's position in the int code . The "Fetch" method must be positioned after the "Execute" method, but this is important only for static SQL, i.e. cursors inside queries.

In general, the following operations are performed within this method:

1. Check whether we've reached the end of data
2. If there is still some data left: Create a new %List and assign a value to the Row variable
3. Otherwise, set AtEnd to 1
4. Prepare qHandle for the next result fetch
5. Return the status

This is how it will look like in our example:

```
ClassMethod AllRecordsFetch(ByRef qHandle As %Binary, ByRef Row As %List, ByRef AtEnd
 As %Integer = 0) As %Status {
    #; Iterating through ^Utils.CustomQueryD
    #; Writing the next id to qHandle and writing the global's value with the new id
into val
    Set qHandle = $Order(^Utils.CustomQueryD(qHandle),1,val)
    #; Checking whether there is any more data left
       If qHandle = "" {
        Set AtEnd = 1
        Set Row = ""
        Quit $$$OK
    }
    #; If not, create %List
    #; val = $Lb("", Prop1, Prop2) see Storage definition
    #; Row =$lb(Id,Prop1, Prop2)  see ROWSPEC for the AllRecords request
    Set Row = $Lb(qHandle, $Lg(val,2), $Lg(val,3))
    Quit $$$OK
}
```

The queryNameClose method

The queryNameClose method terminates the query, once all the data is obtained. The signature of this method is as follows:

```
ClassMethod queryNameClose(ByRef qHandle As %Binary) As %Status [ PlaceAfter = queryN
ameFetch ]
```

where:

- Caché executes this method after the final call to the queryNameFetch method
- In other words, this is a query destructor
- Therefore you should dispose all SQL cursors, queries and local variables in its implementation
- The methods return the current status

In our example, we have to delete the local variable qHandle:

```
ClassMethod AllRecordsClose(ByRef qHandle As %Binary) As %Status {
    Kill qHandle
    Quit $$$OK
  }
```

And here we are! Once you compile the class, you will be able to use the AllRecords query from %SQL.Statement – just as the basic class queries.

Iteration logic approaches for custom queries

So, what approaches can be used for custom queries? In general, there exist 3 basic approaches:

- Iteration through a global
- Static SQL
- Dynamic SQL

Iteration through a global
The approach is based on using $Order and similar functions for iteration through a global. It can be used in the following cases:

- Data is stored in globals (without classes)
- You want to reduce the number of glorefs is the code
- The results must be/can be sorted by the global's subscript

Static SQL
The approach is based on cursors and static SQL. This is used for:

- Making the int code more readable
- Making the work with cursors easier
- Speeding up the compilation process (static SQL is included into the class query and is therefore compiled only once).

Note:

- Cursors generated from queries of the %SQLQuery type are named automatically, e.g. Q14.
- All cursors used within a class must have different names.
- Error messages are related to the internal names of cursors which have an additional characters at the end of their names. For example, an error in cursor Q140 is actually caused by cursor Q14.
- Use PlaceAfter and make sure that cursors are used in the same int routinewhere they have been declared.
- INTO must be used in conjunction with FETCH, but not DECLARE.

Example of static SQL for Utils.CustomQuery:

```
Query AllStatic() As %Query(CONTAINID = 1, ROWSPEC = "Id:%String,Prop1:%String,Prop2:
%Integer") [ SqlName = AllStatic, SqlProc ]
{
}

ClassMethod AllStaticExecute(ByRef qHandle As %Binary) As %Status
{
    &sql(DECLARE C CURSOR FOR
        SELECT Id, Prop1, Prop2
        FROM Utils.CustomQuery
     )
     &sql(OPEN C)
    Quit $$$OK
}

ClassMethod AllStaticFetch(ByRef qHandle As %Binary, ByRef Row As %List, ByRef AtEnd
As %Integer = 0) As %Status [ PlaceAfter = AllStaticExecute ]
{
    #; INTO must be with FETCH
    &sql(FETCH C INTO :Id, :Prop1, :Prop2)
    #; Check if we reached end of data
    If (SQLCODE'=0) {
        Set AtEnd = 1
        Set Row = ""
        Quit $$$OK
    }
    Set Row = $Lb(Id, Prop1, Prop2)
    Quit $$$OK
}

ClassMethod AllStaticClose(ByRef qHandle As %Binary) As %Status [ PlaceAfter = AllSta
ticFetch ]
{
    &sql(CLOSE C)
    Quit $$$OK
}
```

Dynamic SQL

The approach is based on other class queries and dynamic SQL. This is reasonable when in addition to an SQL query itself, you also need to perform some additional operations, e.g. execute an SQL query in several namespaces or escalate permissions before running the query.

Example of dynamic SQL for Utils.CustomQuery:

```
Query AllDynamic() As %Query(CONTAINID = 1, ROWSPEC = "Id:%String,Prop1:%String,Prop2
:%Integer") [ SqlName = AllDynamic, SqlProc ]
{
}

ClassMethod AllDynamicExecute(ByRef qHandle As %Binary) As %Status
{
    Set qHandle = ##class(%SQL.Statement).%ExecDirect(,"SELECT * FROM Utils.CustomQue
ry")
    Quit $$$OK
}

ClassMethod AllDynamicFetch(ByRef qHandle As %Binary, ByRef Row As %List, ByRef AtEnd
 As %Integer = 0) As %Status
```

```
{
    If qHandle.%Next()=0 {
        Set AtEnd = 1
        Set Row = ""
        Quit $$$OK
    }
    Set Row = $Lb(qHandle.%Get("Id"), qHandle.%Get("Prop1"), qHandle.%Get("Prop2"))
    Quit $$$OK
}

ClassMethod AllDynamicClose(ByRef qHandle As %Binary) As %Status
{
    Kill qHandle
    Quit $$$OK
}
```

Alternative approach: %SQL.CustomResultSet

Alternatively, you can create a query by subclassing from the %SQL.CustomResultSet class. Benefits of this approach are as follows:

- Slight increase in speed
- ROWSPEC is unnecessary, since all metadata is obtained from the class definition
- Compliance with the object-oriented design principles

To create query from the subclass of %SQL.CustomResultSet class, make sure to perform the following steps:

1. Define the properties corresponding to the resulting fields
2. Define the private properties where the query context will be stored
3. Override the %OpenCursor method (similar to queryNameExecute) which initiates the context. In case of any errors, set %SQLCODE and %Message as well
4. Override the %Next method (similar to queryNameFetch) which obtains the next result. Fill in the properties. The method returns 0 if all the data has been processed and 1 if some data is still remaining
5. Override the %CloseCursor method (similar to queryNameClose) if necessary

Example of %SQL.CustomResultSet for Utils.CustomQuery:

```
Class Utils.CustomQueryRS Extends %SQL.CustomResultSet
{
Property Id As %String;
Property Prop1 As %String;
Property Prop2 As %Integer;
Method %OpenCursor() As %Library.Status
{
    Set ..Id = ""
    Quit $$$OK
}

Method %Next(ByRef sc As %Library.Status) As %Library.Integer [ PlaceAfter = %Execute
 ]
{
    Set sc = $$$OK
    Set ..Id = $Order(^Utils.CustomQueryD(..Id),1,val)
    Quit:..Id="" 0
```

```
    Set ..Prop1 = $Lg(val,2)
    Set ..Prop2 = $Lg(val,3)
    Quit $$$OK
}
}
```

You can call it from Caché Object Script code in the following way:

```
 Set resultset= ##class(Utils.CustomQueryRS).%New()
       While resultset.%Next() {
        Write resultset.Id,!
 }
```

Another example is available in the SAMPLES namespace – it's the Sample.CustomResultSet class which implements a query for Samples.Person.

Summary

Custom queries will help you to separate SQL expressions from Caché Object Script code and implement sophisticated behavior which can be too difficult for pure SQL.

References

Class queries

Iteration through a global

Static SQL

Dynamic SQL

%SQL.CustomResultSet

The Utils.CustomQuery class

The Utils.CustomQueryRS class

The author would like to thank Alexander Koblov for his assistance in writing this article.

#Best Practices #Compiler #Languages #Object Data Model #ObjectScript #SQL #Caché