Calling 4D Methods From SQL Statements

By Atanas Atanassov, Technical Services Team Member, 4D Inc.

Technical Note 09-37

Table of Contents

Abstract

Calling 4D project methods from SQL statements allows developers to use 4D code that already works to improve their searches. This option does not alter the structure of the SQL statement and any project method can be called from within SQL. In this way developers can have the benefit of reusing 4D code from within a SQL statement to perform more complex tasks. Developers more familiar with SQL can implement the functionality of 4D language without worrying about syntax details so long as they understand the method's input and expected output.

Introduction

Calling a 4D method from inside a SQL statement is an option that makes SQL queries very flexible. It allows developers familiar with SQL to use 4D methods inside their queries and provides a convenient way to use existing methods. For example, developers can take advantage of a query across multiple tables using 4D's relations without having to construct a join statement while still using SQL.

This Technical Note explains how to make calls to methods from inside SQL statements. To be able to understand the examples, the reader should have basic understanding of SQL. This is not a SQL guide and does not explain the language syntax, nor does not show how to build SQL queries. The purpose of this Technical Note is to demonstrate how to call 4D methods from within a SQL statement.

How to call a 4D method in a SQL statement

First, only project methods can be called from a SQL statement. The project method can have input parameters and return values.

Observe the following example:

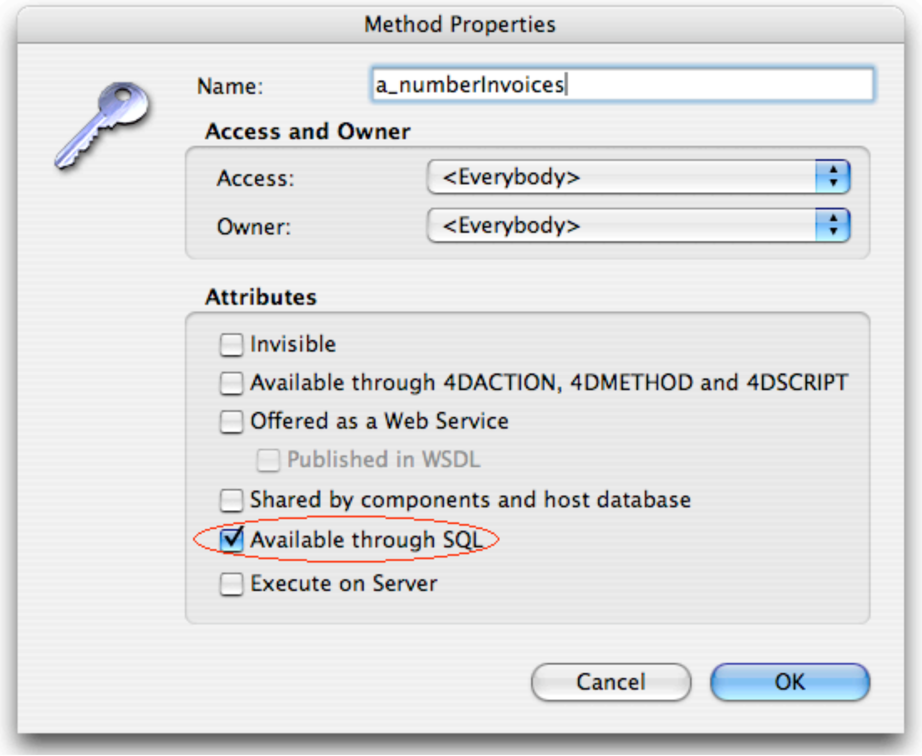
```
ARRAY TEXT($arr;0)

Begin SQL
  SELECT ITEM
  FROM Product
  WHERE Unit_Price<5
  INTO :$arr
End SQL
```

This is a simple SQL query that finds Items that have a Unit_Price of less than $5.00. This value is stored into the $arr array. We have not introduced calling a method yet, but take note of what this code does when it executes.

Assume you have a project method that performs the equivalent query in the 4D language. Instead of writing out new code that accomplishes the same task, you can simply call that method to execute and return the results.

First, to be able to execute the called method, the “Available through SQL” option in the method’s properties should be checked:


Method Properties

Name: a_numberInvoices

Access and Owner

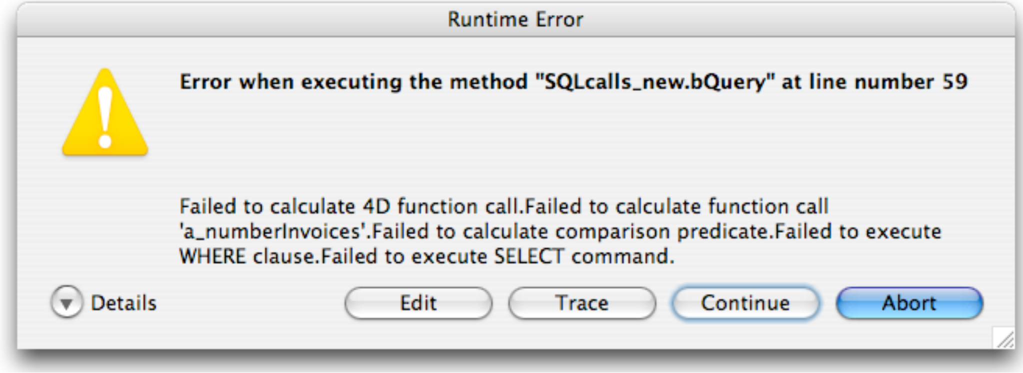
Access: <Everybody>

Owner: <Everybody>

Attributes

☐ Invisible
☐ Available through 4DACTION, 4DMETHOD and 4DSCRIPT
☐ Offered as a Web Service
  ☐ Published in WSDL
☐ Shared by components and host database
☑ Available through SQL
☐ Execute on Server

For security reasons this option is unchecked by default. When it is unchecked and there is a SQL call to this method, an error similar to the following is returned.


Runtime Error

Error when executing the method "SQLcalls_new.bQuery" at line number 59

Failed to calculate 4D function call.Failed to calculate function call 'a_numberInvoices'.Failed to calculate comparison predicate.Failed to execute WHERE clause.Failed to execute SELECT command.

The contents of the called method are as follows. Note that it is an equivalent query to the SQL code above:

```
`Method: method_called
`----------------------

C_TEXT($1;$product;$0)

$product:=$1
Query([Products];[Products]ITEM=$product)
```

```
If([Products]Units_Price<5)
   $0:=[Products]ITEM
End If
```

Note that in the method, method_called, we are still looking for products that cost less than $5.00. The results of both queries are almost the same. Why almost? We will talk about this in a bit. For now, notice how our calling method has changed:

```
ARRAY TEXT($arr;0)

Begin SQL
  SELECT {fn method_called(ITEM) AS VARCHAR)
  FROM Product
  INTO :$arr
End SQL
```

In this method, we specify that we are going to call a method in the SELECT statement by naming the method, method_called, in curly braces. The “fn” means that this method can be used as a function, and can have parameters and a return value. The parameters are listed in the parentheses after the method name. In this case, we are only passing Item as a parameter. The type of the method’s return value is specified in AS clause. In this example the type is VARCHAR.

Depending on the method call, the return value can be used in the SQL statement or sent as a result to the INTO clause. In our example the return value is sent to the array called $arr, just as it was in the very first example.

Note: *Mismatching the SQL type in the AS clause and the type of the method’s return value will fail the query. The elements of the array will be populated with null values.*

Available return types are all data SQL types supported in 4D.

Because of the way SELECT works, “method_called” is called for every record in Products table, and all return values are be sent to the result array, including the ones that are greater or equal to $5.00. Thus, the result array $arr has the same number of elements as the number of records. Array elements for the records not matching the query criteria are set to 0.

This differs from the array from the very first example that has only the records matching the query, not all records. In order to fix this, we need to change the code in second example by including the WHERE clause:

```
ARRAY TEXT($arr;0)

Begin SQL
  SELECT {fn method_called(ITEM) AS VARCHAR)
  FROM Product
  WHERE {fn method_called(ITEM) AS VARCHAR}>’’
  INTO :$arr
End SQL
```

Now both arrays have the same number of elements.

This example demonstrates that including 4D methods in SQL statements can produce different results, but there is a way to fix that.

Passing parameters to methods called in SQL statements

Passing parameters to a method called from SQL is different than passing parameters to a method from the 4D language. In 4D, the standard parameter delimiter for parameter list is a semicolon “;”.

Assume that the method called “testMethod” has two parameters $var1 and $var2. The call to this method in 4D is simply:

```
testMethod($var1; $var2)
```

When this method is called in a SQL statement, the parameter delimiter is a comma. Going back to the original example, the “method_called” method has a hard-coded price in the IF statement ($5.00). In order to make this method more generic, we can pass the price as second argument in the method call.

```
`Method: method_called

C_TEXT($1;$product;$0)
C_REAL($2;$pice)

$product:=$1
$price:=$2

QUERY([Products];[Products]ITEM=$product)
If([Products]Unit_Price<$price)
   $0:=[Products]Utit_Price<$price
Else
    $0:=””
End if
```

The SQL statement changes to:

```
ARRAY TEXT($arr;0)
C_Real($unitPrice)

$unitPrice:=5

Begin SQL
  SELECT {fn method_called(ITEM,:$unitPrice) AS VARCHAR)
  FROM Product
  WHERE {fn method_called(ITEM,:$unitPrice) AS VARCHAR}>’’
  INTO :$arr
End SQL
```

We are now pasing both the Item column and the $unitPrice variable to method_called using a comma to separate the two. Note the colon that preceeds $unitPrice. Passing a variable can also be done by adding “<<>>” around variable name and removing the colon like so:

```
ARRAY TEXT($arr;0)
C_Real($unitPrice)

$unitPrice:=5

Begin SQL
  SELECT {fn method_called(ITEM,<<$unitPrice>>) AS VARCHAR}
  FROM Product
  WHERE {fn method_called(ITEM,<<$unitPrice>>) AS VARCHAR}>’’
  INTO :$arr
End SQL
```

Typing semicolon or missing the colon in front of the 4D variables will cause parsing errors. Variable and method names can be only 32 characters long.

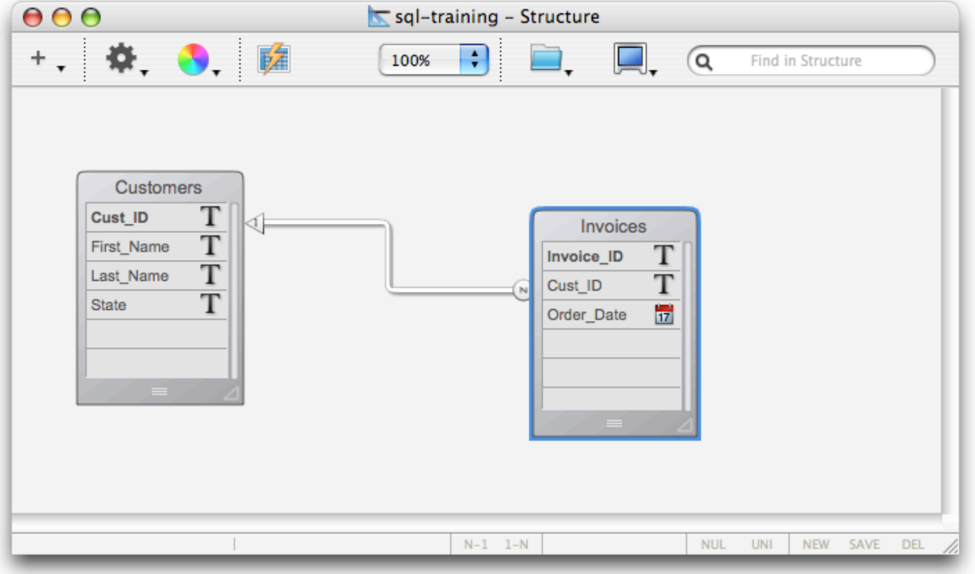
Returning value from a 4D method called inside SQL statement is stored in $0. The type of returning value should match the type specified in AS clause in the SQL statement.

SQL calls in 4D – overview and demo

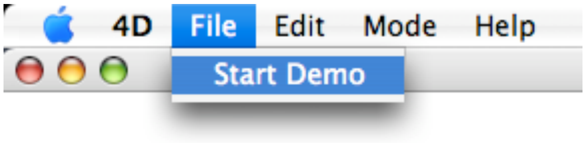
SQL statements can be executed with ***Begin/End SQL***, ***SQL Execute***, ***and Execute immediate***.

The sample database demonstrates calling a 4D method from SQL using these ways. The point of the sample database is to demonstrate how to call 4D methods inside SQL statements. It is not meant to be a SQL guide or to teach the reader how to build SQL queries.

The structure consists of two tables: “Customer” and “Invoices”. There is an automatic relation between both tables:

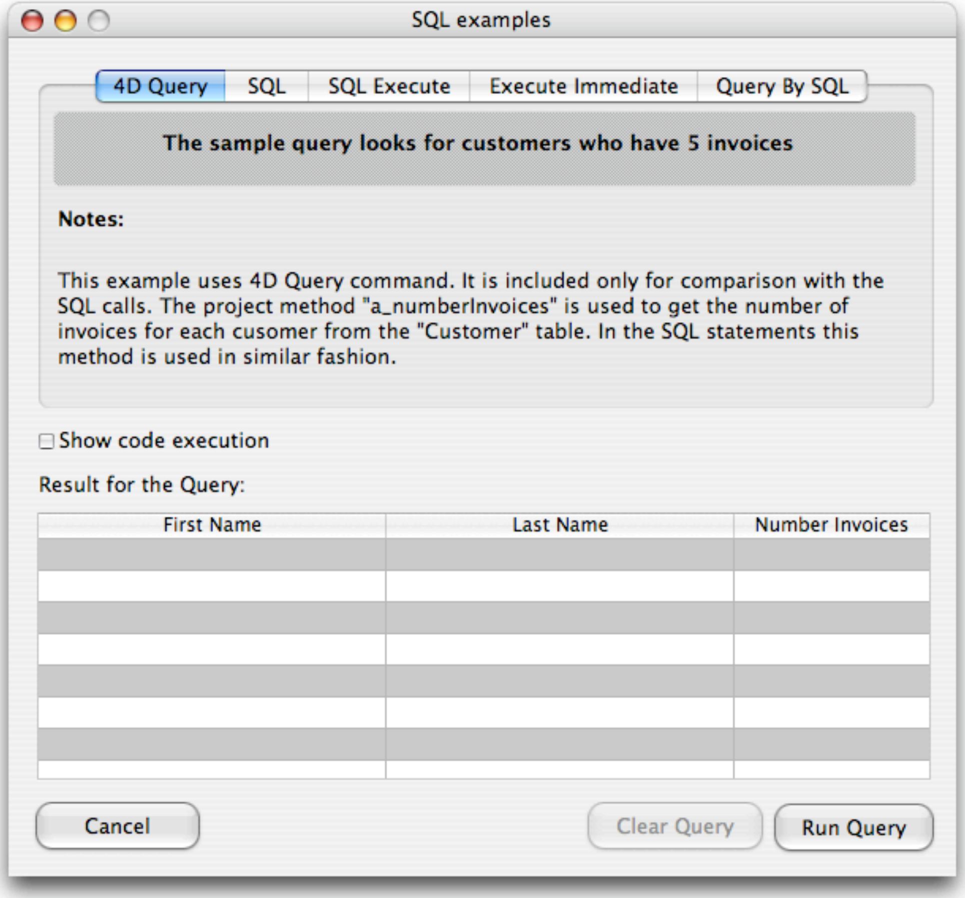


With the database open, select File->Start Demo to start the demo.



Once the demo is launched, all examples can be seen in a form called "SQL examples".

- Form tabs are named after the SQL call examples. For example, the SQL EXECUTE command is demonstrated in the SQL Execute tab and so on.
- To be able to run the query, the user needs click on "Run Query" button.
- The "Show code execution" option allows to the user to see the code inside the debugger window.



The sample query looks for customers who have 5 invoices

Notes:

This example uses 4D Query command. It is included only for comparison with the SQL calls. The project method "a_numberInvoices" is used to get the number of invoices for each cusomer from the "Customer" table. In the SQL statements this method is used in similar fashion.

☐ Show code execution

Result for the Query:

First Name	Last Name	Number Invoices

The SQL Tab

Begin SQL and End SQL phrases encloses the SQL statement to be executed. This format is identical to the examples given earlier in this document. In the sample database, the SQL statement is as follows:

```
Begin SQL
   SELECT First_Name,Last_Name,{fn a_numberInvoices(Cust_ID) AS NUMERIC}
   FROM Customers
   WHERE {fn a_numberInvoices(Cust_ID) AS NUMERIC}=:$invoiceNumber
   INTO :firstName_a,:lastName_a, :numbInv_a;
End SQL
```

In this SQL statement, we select the First_Name and the Last_Name fields from the “Customers” table. “a_numberInvoices” is the 4D project method to be called. Cust_ID is the parameter that is passed and also happens to be the name of the Foreign Key in “Invoices” table. The return value is the number of invoices per customer.

Because this method is included in the SELECT clause, it is called for every record in “Customers” table. This is very helpful if we want to keep track for total number of invoices for every customer. This method is also included in the WHERE clause, and the return value is compared with the $invoiceNumber variable, which is equal to 5 here.

The code for “a_numberInvoices” is;

```
C_TEXT($1;$customerID)
C_LONGINT($0)

$customerID:=$1
QUERY([Invoices];[Invoices]Cust_ID=$customerID)
$0:=Records in selection([Invoices])
```

The Execute Immediate Tab

Execute immediate is a 4D command that allows developers to build dynamic queries. First the SQL statement should be built and saved into a variable and then executed with the command.

```
`Create the SQL statement

C_TEXT($QText)

$Qtext:=""
$Qtext:=$Qtext+"SELECT First_Name,Last_Name,{fn a_numberInvoices(Cust_ID) AS
NUMERIC}"
$Qtext:=$Qtext+" FROM Customers"
$Qtext:=$Qtext+" WHERE {fn a_numberInvoices(Cust_ID) AS
NUMERIC}=:$invoiceNumber"
$Qtext:=$Qtext+" INTO :firstName_a,:lastName_a,:numbInv_a;"

Begin SQL
   EXECUTE IMMEDIATE :$Qtext
End SQL
```

In this example, we use a hard coded query that is similar to the previous example. Rest assured that calling a 4D method from within a SQL statement can be done dynamically using EXECUTE IMMEDIATE, as it is done here.

The SQL Execute Tab

This command also is used for building dynamic queries. First the user needs to login into the database and then execute the query.

```
SQL LOGIN(SQL_INTERNAL ;"";"")
If (OK=1)
   $Qtext:=""
   $Qtext:=$Qtext+"SELECT First_Name,Last_Name,{fn a_numberInvoices(Cust_ID) AS
NUMERIC}"
   $Qtext:=$Qtext+" FROM Customers"
   $Qtext:=$Qtext+" WHERE {fn a_numberInvoices(Cust_ID) AS
NUMERIC}=:$invoiceNumber"
   SQL EXECUTE($Qtext;firstName_a;lastName_a;numbInv_a)
   SQL LOAD RECORD(SQL All Records )
   SQL LOGOUT
End if
```

QUERY BY SQL

Query By SQL uses the SQL engine to perform the query. The second parameter is the WHERE clause from the SQL statement. The "a_numberInvoices" method is called to compare the return value with $invoiceNumber, which is equal to 5 in this example. After executing this command, it creates a selection and the current record is the first record from this selection. A for loop goes through the records in this selection.

```
C_TEXT($Qtext)

$loopIndex:=0
$arrIndex:=0
$Qtext:=" {fn a_numberInvoices(Cust_ID) AS NUMERIC}=:$invoiceNumber"
QUERY BY SQL([Customers];$Qtext)
   For ($loopindex;1;Records in selection([Customers]))
    $invoces:=a_numberInvoices ([Customers]Cust_ID)
      If ($invoces>=$invoiceNumber)
          $arrIndex:=$arrIndex+1
          INSERT IN ARRAY(firstName_a;$arrIndex;1)
          firstName_a{$arrIndex}:=[Customers]First_Name
          INSERT IN ARRAY(lastName_a;$arrIndex;1)
          lastName_a{$arrIndex}:=[Customers]Last_Name
          INSERT IN ARRAY(numbInv_a;$arrIndex;1)
          numbInv_a{$arrIndex}:=$invoces
       End if
       NEXT RECORD([Customers])

   End for
```

Conclusion

Calling 4D methods in SQL statement allows 4D developers to include 4D code in their SQL statements. On the other site allows SQL users to use 4D methods and make their queries more flexible.