Translating SQL from Access to SQL Server

In the first of two articles, Andrew Couch looks into issues surrounding the basic conversion of SQL, and examines tactical approaches for dealing with Access product-specific features which have been incorporated into the SQL

BY ANDREW COUCH

When faced with the task of converting an Access database containing several hundred queries to SQL Server, you need a structured approach to tackle what is a sizeable problem. Assuming you have already converted the database structure, you could choose to keep your Access application and convert only queries with poor performance or, if you are planning on replacing the Access front-end application, you may need to convert all the queries to Views and Stored Procedures. Achieving a successful conversion will be easier if you have a number of tactical solutions for dealing with conversion issues and you are aware of key differences between the products.

Conversion basics

Before delving into the depths of translating SQL, there are a number of simple differences between Access and SQL Server to be considered. Strings in Access SQL can use either double or single quotes, although the use of double quotes is more common. The double quotes need to be converted to single quotes; for example converting *"Product"* into *'Product'*.

In Access string concatenation can use either + or &. The difference between these is that when the argument is null the + returns the entire expression as NULL, whereas the & only evaluates that part of the expression as an empty string. As a consequence of this, Access developers tend to use & which requires translation to + in SQL Server. The full implication of this is that an expression in Access like [FirstName] & " " & [Surname] should be strictly translated to ISNULL([FirstName],'') + ' ' + ISNULL([Surname],''); otherwise for individuals where only the [Surname] is given the expression will return NULL.

The literal characters used for pattern matching in like clauses must be converted; for example Like "a*" becomes Like 'a%', and each ? changes to _, [!…] changes to [^…] and # changes to [0-9] (whilst more recent JET 4.X versions of Access do support the _ and % characters , traditionally developers have used the ? and * characters in patterns).

Converting to Views or Stored Procedures?

The first tactical decision is in deciding on the corresponding SQL Server objects to be used. If your query is an INSERT, UPDATE, DELETE, MAKE-TABLE or DDL query, then the equivalent object in SQL Server will be a Stored Procedure (Views can not be used in this case). Other types of query such as SELECT or CROSSTAB queries are normally converted to Views (these could also be converted to Stored Procedures if returning read-only data).

Converting a query can be seen as a need to overcome three problems. The first is syntactical conversion, for example removing duplicate columns from an ORDER BY clause, translating wildcards characters "*" to "%" etc, you must make all the changes which are required to allow the converted query to be saved as a SQL Server object. The second problem is establishing that the query executes without generating errors, for example you will need to change the implicit type conversion in Access associated with the concatenation of a number and a string *([ProductId] & [ProductShortName])* to use a SQL Server type conversion function *([ProductId] + CONVERT(VARCHAR([ProductShort-Name]))*. The final problem is ensuring that the data returned is as expected, for example dividing integers gives different results in the two products due to different rounding conventions for integer arithmetic *(in Access 13/3 = 4.3333, in SQL Server 13/3 = 4)*.

Handling Access parameters & references to form controls

Access queries can contain parameters in the SQL, for example [Enter Start Date] and also more complex control references such as [Forms]![frmPickCountry]![cboCountry]. The first example automatically prompts the user when opening the query to enter a date (in Access anything not resolved in a query to the name of a column is assumed to be a parameter), and the second example expects the form called frmPickCountry to be open when the query is executed. Data types in parameter declarations may or may not be included in the Access SQL.

Access – SQL

```
PARAMETERS [Enter Start Date] DateTime;
SELECT Orders.*, Orders.ShipCountry
FROM Orders
WHERE (((Orders.OrderDate)>[Enter
Start Date]) AND
((Orders.ShipCountry)=[Forms]![
frmPickCountry]![cboCountry]));
```

A useful strategy in dealing with these parameterised queries is to construct a table in SQL Server to hold the user's choice of parameters, and convert the SQL to filter by the user's choices. Below we create a parameter table, and supply a view which restricts the results to the active user.

SQL Server – SQL

```
CREATE TABLE UserParameters(UserId
INT IDENTITY(1,1) PRIMARY KEY,
UsernameFilter VARCHAR(100) DEFAULT
suser_name(), param_StartDate
DATETIME,param_Country VARCHAR(50) )
CREATE VIEW vw_UserParameters AS
SELECT * FROM UserParameters
WHERE UsernameFilter = suser_name()
```

The SQL would now be converted as follows:-

```
SELECT Orders.*, Orders.ShipCountry
FROM Orders CROSS JOIN
vw_UserParameters p WHERE
(((Orders.OrderDate)>p.[
```

```
param_StartDate]) AND
((Orders.ShipCountry)=p.[param_Country]))
```

This can be taken one step further in making the parameters optional using COALESCE functions and ensuring parameters are set to NULL when not required. (Notice that to avoid testing for NULL = NULL we provide the value 'N/A' to make the comparison 'N/A' = 'N/A', this would change depending on the column data type to 0 = 0 for an integer etc.).

```
SELECT Orders.*, Orders.ShipCountry
FROM Orders CROSS JOIN vw_UserParameters
p WHERE (((Orders.OrderDate)>p.[
param_StartDate]) AND (COALESCE(
Orders.ShipCountry, 'n/a')=
COALESCE(p.[param_Country],
Orders.ShipCountry,'n/a')))
```

When a user makes choices in your application you need to update the user's choices into the parameter view (setting any unselected columns to null) vw_UserParameters, and then execute the appropriate view or stored procedure.

An alternative technique when using a stored procedure is to replace the Access parameter with SQL Server Variables (although there is no equivalence here when dealing with Views).

```
CREATE PROC OrdersForCountry(
@StartDate DATETIME, @Country
VARCHAR(20)) AS SELECT Orders.*,
Orders.ShipCountry FROM Orders
WHERE ((Orders.OrderDate)>@StartDate)
AND (Orders.ShipCountry=@Country))
```

Handling complex calculations

It is not unusual in applications to find deep layered queries involving complex calculations where use has been made of Access allowing you to create a calculated field which references other calculated fields without restating the calculation; SQL Server does not allow this. In complex applications where calculations can contain nested conditional IIF statements restating all calculations can involve introducing an unacceptable level of additional complexity in your already complex SQL.

One solution is to create a set of views to perform the calculations in several steps, but an alternative is to use nesting in the query to achieve the same effective referencing as available in Access.

Access – SQL

```
SELECT tblRates.[NoOfMonths],
CDbl([NoOfMonths])/1.6
AS [Rate1], CDbl([NoOfYears])/1.6
AS [Rate2], [Rate1]*[Rate2]
AS [Rate3], [Rate3]*[Rate1]
AS [Rate4], [Rate1]*6
AS [Rate1a], [rate1]*6
AS [Rate2a] FROM tblRates;
```

Nested queries, allow the results of a query to be fed into the select of another query, and by moving the calculations into layers in a nested query the reference problems can be resolved without introducing too much additional complexity.

SQL Server – SQL

```
CREATE VIEW
[022_SelfReferenceCalculations]
AS SELECT *,
[Rate3]*[Rate1] AS [Rate4],
[Rate1]*6 AS [Rate1a],
[rate1]*6 AS [Rate2a]
FROM
   (SELECT *,[Rate1]*[Rate2] AS [Rate3]
   FROM (SELECT tblRates.[NoOfMonths],
   CONVERT(FLOAT,[NoOfMonths])/1.6
   AS [Rate1], CONVERT(FLOAT,
   [NoOfYears])/1.6 AS [Rate2]
   FROM tblRates ) AS Nested2
   ) AS Nested1
```

Calculations involving dates

If you have used Access functions for date arithmetic such as the DATEDIFF function then you will find conversion is relatively straightforward, but if you use explicit date arithmetic such as in determining the number of elapsed days using ([StartDate]-[EndDate]) then this will give unexpected results and must be converted to use, for example, the DATEDIFF function; DATEDIFF(d,[EndDate],[StartDate]).

Access – SQL

```
SELECT Orders.[OrderID],
   Orders.[RequiredDate],
   Orders.[ShippedDate],
   [RequiredDate]-[ShippedDate]
   AS [Overdue] FROM Orders;
```

SQL Server – SQL

```
CREATE VIEW
[004_DateArithmetic_Subtraction] AS
SELECT Orders.[OrderID],
Orders.[RequiredDate],
Orders.[ShippedDate],
DateDiff(d,[ShippedDate],
[RequiredDate]) AS [Overdue]
FROM Orders
```

Calculations involving Integer arithmetic

Integer division in Access gives a real number result. Yet in SQL Server it gives an integer result. To obtain similar results in SQL Server explicit type conversions must be introduced. Shown below is an example illustrating that when one part of the calculation is converted real arithmetic is then employed.

Access – SQL

```
SELECT tblRates.[NoOfMonths],
tblRates.[NoOfYears],
[NoOfMonths]/[NoOfYears]
AS [Rate], 13/[NoOfYears]
AS [AltRate1], [NoOfYears]/13
AS [AltRate2] FROM tblRates;
```

SQL Server – SQL

```
CREATE VIEW [021_IntegerDivision] AS
SELECT [tblRates].[NoOfMonths],
[tblRates].[NoOfYears],
CONVERT(REAL,[NoOfMonths])/
[NoOfYears] AS [Rate],13/[NoOfYears]
AS [AltRate1],
CONVERT(REAL,[NoOfYears])/13 AS
[AltRate2] FROM [tblRates]
```

Implicit type conversion in Access is a problem that pervades conversion and you will see further instances where explicit type casting or conversion is required.

Translating functions

There is some degree of equivalence between functions in Access and those available in SQL Server. However, sometimes the parameters will need re-ordering, such as when converting InStr([start,]string1, string2[, compare]) to CHARINDEX (expression1 ,expression2 [, start_location]). Some Access functions allow either additional parameters (which are thankfully not often used), or allow functions accepting fewer parameters than the SQL Server equivalent function.

It is unfortunate that one of the most commonly used functions in Access is the Format function, which is infinitely flexible and as such exceptionally difficult to translate to work with all possible formatting parameters. However, in our experience any given developer tends to have a finite set of favourite formatting strings such as "mm/yyyy" and so this can be accommodated by constructing a SQL Server function to cater for the specific choices in a given application.

Conclusion

This article has described a number of basic issues to be considered when translating SQL, and provided a description of methods for handling both complex calculations and references to screen controls and query parameters. Next time we will look at a wide range of additional issues, including the translation of conditional logic, domain functions and issues relating to date and time data.

■ Andrew Couch is an Office Access MVP, Director of the UK Access User Group and author of MUST, the Migration Upsizing SQL Tool available from **www.upsizing.co.uk**. Andrew also acts as a Consultant and Trainer specialising in Access and SQL Server applications, and can be contacted at **andy@ascassociates.biz**.