Translating SQL from Access to SQL Server, part 2

Converting from Access SQL to SQL Server SQL is a subtle process fraught with danger and difficulties. Andrew Couch shares his experience to guide you through the minefield.

BY ANDREW COUCH

In the first instalment of this two-part article, we looked at both the basics involved in translating SQL and the need to translate SQL, which used complex calculations and references to screen and query parameters. Now we are ready to investigate more localised structural differences in the SQL, and illustrate how to resolve differences.

Conditional logic in SQL

In Access, IIF statements are used for introducing conditional logic into the SQL, a Choose function is also provided for the simple conditional selection of values. The IIF structures can be translated into searched CASE statements which can be nested to match the nesting in Access, and the Choose function translated into a simple CASE structure.

Access – SQL

```
SELECT tblRates.[NoOfMonths],
   tblRates.[NoOfYears],
   IIf([NoOfYears]>5,IIf([NoOfMonths]>10,
   'Over 5 years and over 10 months',
   'Only over 5 years and less than 10
      months'))
   AS [NestedIIF]
FROM tblRates;
```

SQL Server – SQL

```
CREATE VIEW [024_NestedIIF]
AS
SELECT tblRates.[NoOfMonths],
   tblRates.[NoOfYears],
   CASE WHEN [NoOfYears]>5 THEN
      CASE WHEN [NoOfMonths]>10
      THEN 'Over 5 years and over 10
       months'
      ELSE 'Only over 5 years and less
       than 10 months'
      END
   END AS [NestedIIF]
FROM tblRates
```

Access – SQL

```
SELECT DataTable.Number,
   DataTable.MyChoice,
   Choose([MyChoice],"Number 1",
   "Number2", "Number3")
AS TheChoice
FROM DataTable;
```

SQL Server – SQL

```
CREATE VIEW [002_Choice_Function]
AS
SELECT [DataTable].[MyChoice],
   CASE [MyChoice] WHEN 1 THEN 'Number 1'
      WHEN 2 THEN 'Number2'
      WHEN 3 THEN 'Number3'
      ELSE NULL END AS [TheChoice]
   FROM [DataTable]
```

Insert, update and delete

Insert, update and delete queries can't be performed in a View, but require the use of a stored procedure. When converting simple queries the list of any additional fields must be removed and the use of ".*" wildcards removed.

Access – SQL

```
DELETE Customers.*, Customers.[Region]
   FROM Customers
   WHERE (((Customers.[Region])="bc"));
```

SQL Server

```
CREATE PROCEDURE [016_DeleteQuery]
AS
BEGIN
   DELETE
   FROM Customers
   WHERE (((Customers.[Region])='bc'))
END
```

Queries which modify data in a single table but are constrained by additional joins to other tables need additional reformatting. The additional change is in re-stating the table name in the FROM clause, this applies for UPDATE and DELETE queries.

Access – SQL

```
DELETE ControlCustomers.*,
   ControlCustomers.[Region]
   FROM ControlCustomers
      INNER JOIN Customers ON
       ControlCustomers.[CustomerID]=
        Customers.[CustomerID]
      WHERE (((ControlCustomers.[
       Region])="bc"));
```

SQL Server – SQL

```
CREATE PROCEDURE [020_DeleteQueryJoined]
AS
BEGIN
DELETE
FROM ControlCustomers
   INNER JOIN Customers
      ON ControlCustomers.[CustomerID]=
       Customers.[CustomerID]
WHERE (((ControlCustomers.[
   Region])='bc'))
End
```

Additional conversion issues arise for queries which change data in multiple tables, and in such circumstances the query would need to be re-written as a series of multiple steps.

Comparisons involving dates

It is common in Access applications to use the Date() function in a where clause.

Access – SQL

```
WHERE [DateEntered] = Date() AND
   [TimeEntered] = Time()
```

This is not directly equivalent to comparing against the SQL Server function *GETDATE()*, which is a more direct equivalent to the *Now()* Access function. Neither SQL Server nor Access have *DATE* data types for columns, but use a *DATETIME* data type.

Access hides the time information through an interface which always sets a default time when not supplied as "00:00:00", and by default only displays the time when it does not have this value in the field. The *Date* function is designed to match against rows where the time is *"00:00:00"* when using *= Date()*. To make an equivalent comparison in SQL Server we need to strip off the time component from the *GETDATE* function.

SQL Server – SQL

```
WHERE [DateEntered =
   CONVERT([datetime],
      CONVERT(VARCHAR,GETDATE(),1),1)
      AND[TimeEntered] =
       CONVERT([datetime],
      '30 December 1899 ' +
      CONVERT(VARCHAR,GETDATE(),14),14)
```

In Access if you only store time in a *DATETIME* field the date gets hidden by the interface and set to 30th December

1899, which means that any comparisons against the *Time()* function must take this into account.

By default, storing a time without a date in SQL Server results in a default date of 1st January 1900 being saved with the time. The differences here must be taken into account if preserving compatibility between existing and new data.

Comparisons involving Boolean expressions

Booleans in Access have the token values (0, No, False) and (-1, Yes, True), where -1 = True, whereas SQL Server has three values (0), (NULL), (1) where 1 = True. A Yes/No field in Access will by default be False, whereas a Bit field in SQL Server will by default be NULL; a good reason for always specifying a default on all Boolean fields in SQL Server.

Access – SQL

```
SELECT Shippers.* FROM Shippers
   WHERE (((Shippers.[UKOnly])=True))
   OR (((Shippers.[UKOnly])=-1)) OR
   (((Shippers.[UKOnly])=Yes));
```

SQL Server – SQL

```
CREATE VIEW [009_Simple_BooleanTesting]
AS
SELECT Shippers.*
   FROM Shippers
   WHERE (((Shippers.[UKOnly])=1))
   OR (((Shippers.[UKOnly])=1)) OR
   (((Shippers.[UKOnly])=1))
```

Access allows Boolean expressions to be constructed in a variety of structures which are best described as implicit, in that you do not need an explicit comparison against True or False, these cause conversion problems.

Access – SQL

```
SELECT Customers.[CustomerID],
   Customers.[Archived],
   IIf([Archived],"Archived")
   AS [IsArchived] FROM Customers;
```

SQL Server – SQL

```
CREATE VIEW [023_ImpicitBooleanInIIF]
AS
SELECT Customers.[CustomerID],
   Customers.[Archived],
   CASE WHEN [Archived] != 0
      THEN 'Archived' END AS [IsArchived]
FROM Customers
```

A similar problem occurs in IIF statements when we have expressions like ([Archived] AND [OldVersion]), which would be translated as either ([Archived] != 0) And ([OldVersion] != 0) or ([Archived] && [OldVersion]) != 0 using a bitwise comparison.

Sometimes rather than understating the Boolean expression, Access will overstate the expression. In this situation the additional Boolean comparison needs to be removed.

Access – SQL

```
((Abs([TotalAmount]-
   [LesserAmount])>0.01)=1)
```

SQL Server – SQL

```
(Abs([TotalAmount]-[LesserAmount])>0.01)
```

Another example is evaluating a field when it is null, where Access can again use an implicit form of evaluation.

Access – SQL

```
WHERE [Address]
```

SQL Server – SQL

```
WHERE [Address] IS NOT NULL
```

Existence testing

Access allows an existence test to be overstated being evaluated against a Boolean result, and this must be converted.

Access – SQL

```
EXIST(SELECT Manager
   FROM SalesTeam
   WHERE Region = 'Cheshire') = TRUE
```

…and also:

```
EXIST(SELECT Manager
   FROM SalesTeam
   WHERE Region = 'Cheshire') = FALSE
```

SQL Server – SQL

```
EXIST(SELECT Manager
   FROM SalesTeam
   WHERE Region = 'Cheshire')
```

…and also:

```
NOT EXIST(SELECT Manager
   FROM SalesTeam
   WHERE Region = 'Cheshire')
```

Aggregates, Domain functions and subqueries

Domain functions can be converted to sub-queries without too much difficulty, although it is questionable as to whether this is the optimal solution, and in most cases a careful re-think of the query should be undertaken.

A typical example is shown below where we have elected just to display a value from another data source (re-writing using a join would be a more appropriate solution in this case, but here we illustrate a generic approach to this problem).

Access – SQL

```
SELECT Customers.[CustomerID],
   Customers.[CompanyName],
   DCount("*","[ControlCustomers]",
      "[CustomerId] = ''"
      & [CustomerId] & "''")
      AS [InAnotherTable] FROM Customers;
```

SQL Server – SQL

```
CREATE VIEW [028_DomainFunction]
AS
SELECT [Customers].[CustomerID],
   [Customers].[CompanyName],
      (SELECT Count(*)
      FROM [ControlCustomers]
      WHERE [CustomerId] = ''' +
        [CustomerId] + ''')
      AS [InAnotherTable]
FROM [Customers]
```

If present in the where clause we would have the following translation:

Access – SQL

```
SELECT Customers.[CustomerID],
   Customers.[CompanyName],
   DCount("*","[ControlCustomers]",
      "[CustomerId] = ''"
      & [CustomerId] & "''")
      AS [InAnotherTable] FROM Customers
        WHERE (((DCount("*",
        "[ControlCustomers]",
        "[CustomerId] = ''"
        & [CustomerId]
        & "''"))>0));
```

SQL Server – SQL

```
CREATE VIEW
   [029_DomainFunctionWithCriteria]
AS
SELECT [Customers].[CustomerID],
   [Customers].[CompanyName],
      (SELECT Count(*)
FROM [ControlCustomers]
WHERE [CustomerId] = ''' + [CustomerId] +
   ''') AS [InAnotherTable]
FROM [Customers]
WHERE ((((SELECT Count(*)
   FROM [ControlCustomers]
      WHERE [CustomerId] = ''' +
        [CustomerId] + ''') )>0))
```

A final example would be one in which the domain function is used as part of a grouping:

Access – SQL

```
SELECT DCount("*","[ControlCustomers]",
   "[CustomerId] = ''"
   & [CustomerId]
   & "''")
      AS [InAnotherTable],
        Count(Customers.[CustomerID])
        AS [CountOfCustomerID]
        FROM Customers GROUP BY
        DCount("*","[ControlCustomers]",
        "[CustomerId] = ''"
        & [CustomerId] & "''");
```

SQL Server – SQL

```
CREATE VIEW
   [030_DomainFunctionWithGrouping]
AS
SELECT (SELECT Count(*)
FROM [ControlCustomers]
   WHERE [CustomerId] = ''' +
      [CustomerId] + ''')
      AS [InAnotherTable],
        Count(Customers.[CustomerID])
      AS [CountOfCustomerID]
FROM [Customers]
   GROUP BY (SELECT Count(*)
FROM [ControlCustomers]
   WHERE [CustomerId] = ''' +
      [CustomerId] + ''')
```

Unfortunately this will give the following error: *"Cannot use an aggregate or a subquery in an expression used for the group by list of a GROUP BY clause."* In this case the GROUP BY must be removed.

It is also possible to translate DOMAIN functions into SQL Server functions (writing your own equivalent function). If doing this then you need to watch that you do not run into additional problems with the data types being returned by the functions. Domain functions in Access can return different types of data, and writing a SQL Server function to return SQL_VARIANT is a solution which can impact other aspects of your SQL; for example you can't perform arithmetic on this data type. This would mean that several functions would be required when implementing a DLOOKUP but only one is likely to be required for a DCOUNT. Performance could also become an issue with a solution utilising functions.

Summary Queries

Access allows a direct application of an aggregate on a Boolean operator (and also on date arithmetic). For example, COUNT([InvoicePaid]) where invoice paid is a Boolean. This requires conversion. I have also come across the use of expressions like MIN([InvoicedPaid]) and MAX([InvoicePaid]).

Access – SQL

```
SELECT Orders.[CustomerID],
   Avg([RequiredDate]-[ShippedDate])
AS [Overdue] FROM Orders
   GROUP BY [Orders].[CustomerID];
```

SQL Server – SQL

```
CREATE VIEW
   [005_Aggregate_OnDateSubtraction]
AS
SELECT Orders.[CustomerID],
   Avg(CONVERT(INT,DateDiff(d,
      [ShippedDate],
      [RequiredDate]))) AS [Overdue]
FROM Orders
   GROUP BY [Orders].[CustomerID]
```

Access – SQL

```
SELECT - Sum(Shippers.[UKOnly])
   AS [SumOfUKOnly] FROM Shippers;
```

SQL Server – SQL

```
CREATE VIEW [007_AggregateOnBoolean]
AS
SELECT Sum( CONVERT(
   INT,Shippers.[UKOnly]))
   AS [SumOfUKOnly]
FROM Shippers
```

Note the removal of the – sign in the above, when data was migrated (-1) values are automatically translated to (1).

If using MIN or MAX, then it must be remembered that True (-1) in Access is (1) in SQL Server.

> "The greatest remaining challenges to any automated conversion are in resolving the implicit type conversions of Access and the implicit Boolean evaluations."

Group By restrictions

It is common practise in Access to overstate a Group By, including unnecessary fields; this is because when a column is entered in the query grid for a query containing totals to display a string such as "Year", this is also automatically added into the group by:

Access – SQL

```
SELECT 1 AS TheNumber, "Code"
   AS TheString, Left([Country],3)
   AS ShortCode,
      Count(Customers.CustomerID)
   AS CountOfCustomerID
FROM Customers
   GROUP BY 1 , "Code",
      Left([Country],3);
```

Upsizing this structure will result in the following error message: *"Each GROUP BY expression must contain at least one column that is not an outer reference."*

This error message is misleading, and will occur if you try and group by any literal values, custom functions or built in functions such as GetDate() (which is probably the most common problem). Such fields must be removed from a grouping:

SQL Server – SQL

```
SELECT 1 AS [TheNumber],
   'Code' AS [TheString],
   LEFT([Country],3) AS [ShortCode],
   Count(Customers.[CustomerID]) AS
      [CountOfCustomerID]
FROM [Customers]
   GROUP BY LEFT([Country],3)
```

In is also possible in Access to edit the SQL removing the additional grouping, this will be remembered by Access although not distinguished in the query grid as having been removed.

Access – SQL

```
SELECT 1 AS TheNumber, "Code"
   AS TheString, Left([Country],3)
   AS ShortCode,
      Count(Customers.CustomerID)
   AS CountOfCustomerID
FROM Customers
   GROUP BY Left([Country],3);
```

Order By restrictions

In a similar manner to Group By, Order By has a similar restriction, which is also due to how Access allows the graphical construction of queries. In this case, however, entering a literal for a grouping which is a numeric does not cause a problem, but a literal string will cause the problem: *"A constant expression was encountered in the ORDER BY list, position 2."*

Access – SQL

```
SELECT 1 AS [TheNumber],
   'Code' AS [TheString],
   LEFT([Country],3) AS [ShortCode],
   Count(Customers.[CustomerID]) AS
      [CountOfCustomerID]
FROM [Customers]
GROUP BY LEFT([Country],3)
   ORDER BY 1,'Code',[TheString],
      LEFT([Country],3)
```

Removing the literal string 'Code' but leaving 1 (which is also a constant removes the error).

Column selections, repetition and clashes

The repetition of a field from a table such as shown below is not allowed.

Access – SQL

```
SELECT Customers.[CustomerID],
   Customers.[CompanyName],
   Customers.[Region],
   Customers.[Region]
FROM Customers
   WHERE (((Customers.[Region])="bc"));
```

SQL Server – SQL

```
CREATE VIEW [014_Fields_DuplicateFields]
AS
SELECT Customers.[CustomerID],
   Customers.[CompanyName],
   Customers.[Region] AS Expr1001 ,
   Customers.[Region]
FROM Customers
   WHERE (((Customers.[Region])='bc'))
```

When joining two tables and selecting two fields with the same name an explicit change of name is required when using a View.

Access – SQL

```
SELECT Customers.[CustomerID],
   OldCustomers.[CustomerID]
FROM Customers INNER JOIN OldCustomers ON
```

```
   Customers.[CustomerID]=
      [OldCustomers].[CustomerID];
```

SQL Server – SQL

```
CREATE VIEW [027_ThreePartStep1b]
AS
SELECT [Customers].[CustomerID]
   AS [Customers.CustomerID] ,
      [OldCustomers].[CustomerID] AS
       [OldCustomers.CustomerID]
FROM [Customers]
   INNER JOIN [OldCustomers]
      ON [Customers].[CustomerID]=
      [OldCustomers].[CustomerID]
```

This becomes a little more complex when we look at a query which uses the above query. Access now uses a three-part reference to the field, which requires adaptation to a two-part reference.

Access – SQL

```
SELECT [027_ThreePartStep1b
   ].Customers.CustomerID,
   [027_ThreePartStep1b].
   OldCustomers.CustomerID
FROM 027_ThreePartStep1b;
```

SQL Server

```
CREATE VIEW [027_ThreePartStep2b]
AS
SELECT [027_ThreePartStep1b].[
   Customers.CustomerID],
   [027_ThreePartStep1b].[
   OldCustomers.CustomerID]
FROM [027_ThreePartStep1b]
```

If wildcard selection of tables is used then this becomes more complex, and the wildcard needs expanding to list all the fields in each source of data.

Ordering data in a VIEW

SQL Server Views are not allowed to contain the ORDER BY clause. However there is a workaround for this (which is used in Microsoft's SSMA tool).

Access – SQL

```
SELECT Customers.[CompanyName]
   FROM Customers
   ORDER BY Customers.[CompanyName] DESC;
```

SQL Server – SQL

```
CREATE VIEW [026_SimpleOrderBy]
AS
SELECT TOP 9223372036854775807 WITH TIES
   [Customers].[CompanyName]
FROM [Customers]
ORDER BY [Customers].[CompanyName] DESC
```

For SQL Server 2000 this can be changed to:

```
SELECT TOP 100% WITH TIES
```

Conclusion

Translating SQL between Access and SQL Server is a challenge because both products are extremely flexible and powerful and have been separately developed with different idiosyncrasies and strengths that developers have exploited.

Once the systematics described in these two articles have been investigated, probably the greatest remaining challenges to any automated conversion are in resolving the implicit type conversions of Access and the implicit Boolean evaluations, which in SQL Server need to be converted into explicit data type conversions and Boolean comparisons.

Access databases tend to have additional complications, such as containing queries which no longer work, and queries which are no longer used in an application. The tidying up of this in addition to the resolution of technical conversion issues remains a challenge to all developers upsizing Access applications.

■ 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**.