Make The dynamic SQL easy to read and debug with REPLACE function

Posted on 2017-08-06 by Jonas Henriksson

Dynamic SQL is by nature hard to read, first you lose the color coding in SSMS, since the it's just a string it'll format as a string, with all text in same color. On top of that the SQL string will be built from different part of meta data so there will be concatenating going on. Add some formatting such as carriage returns and indentions and the readability will be further impaired.

The SQL statement for declaring a foreign key has many meta data dependent parts so it serves as a good example of how ugly it can get. In this post, I use a bare minimum of the foreign key syntax. A fuller syntax will be provided in later post:

```
SELECT        CONVERT(varchar(max), 'ALTER TABLE ' + QUOTENAME(cs.name) + '.' + QUOTENAME(ptab.name) +
            + CONVERT(varchar(max), 'ADD CONSTRAINT ' + QUOTENAME(fk.name) + CHAR(13) + CHAR(10))
            + CONVERT(varchar(max), 'FOREIGN KEY (' )
            + STUFF(( SELECT     ',' + QUOTENAME(c.name)
                      FROM       sys.columns                c
                      INNER JOIN sys.foreign_key_columns    fkc
                             ON  c.column_id              = fkc.parent_column_id
                             AND c.object_id              = fkc.parent_object_id
                      WHERE      fkc.constraint_object_id = fk.object_id
                      ORDER BY   fkc.constraint_column_id
                      FOR XML PATH(''), TYPE).value('.', 'varchar(max)'), 1, 1, '')+ CONVERT(varchar(m
          + CONVERT(varchar(max), 'REFERENCES ' + QUOTENAME(rs.name) + '.' + QUOTENAME(rtab.name) + '
          + STUFF(( SELECT     ',' + QUOTENAME(c.name)
                    FROM       sys.columns                c
                    INNER JOIN sys.foreign_key_columns    fkc
                           ON  c.column_id              = fkc.referenced_column_id
                           AND c.object_id              = fkc.referenced_object_id
                    WHERE      fkc.constraint_object_id = fk.object_id
                    ORDER BY   fkc.constraint_column_id
                    FOR XML PATH(''), TYPE).value('.', 'varchar(max)'), 1, 1, '') + CONVERT(varchar(m
FROM         sys.foreign_keys           fk
INNER JOIN   sys.tables                 rtab
       ON    fk.referenced_object_id =  rtab.object_id
INNER JOIN   sys.schemas                rs
       ON    rtab.schema_id          =  rs.schema_id
INNER JOIN   sys.tables                 ptab
       ON    fk.parent_object_id     =  ptab.object_id
INNER JOIN   sys.schemas                cs
       ON    ptab.schema_id          =  cs.schema_id
```

Did I mention it's ugly? Why do the expression ta create a foreign key get complex? When writing SQL-expression You can be sure there are some implicit data conversion going on. The rules for implicit conversion has surprised my quite often and I ended up with a truncated SQL string. As my solutions became more complex it became difficult to identify where the SQL string got truncated. My solution to this was a brute force approach to convert every part of the concatenation explicitly to varchar(max), hence the frequent use of CONVERT function. The solution was fool proof but not elegant.

After a couple of years, I was a master in writing, not to mention debugging, complex string concatenations. Then it dawned on me. There is a much simpler way to do this, avoiding both implicit conversion and carriage returns.

REPLACE function comes to our aid

The syntax for the REPLACE function is:

```
REPLACE(<SQL template String>, <Placeholder>, <Replacement value>)
```

If the <Placeholder> is found one or more times the <SQL template String> it's replaced with the <Replacement value>. The beauty of the REPLACE function is that the returned data type is that of the first parameter. Hence if <SQL template String> is of the varchar(max) data type so will the return value. This means that your SQL string will not be truncated. If you have more than one replacement value, and you will, you just nest the REPLACE function and the returned data type is still varchar(max).

Furthermore, the carriage returns can be embedded in the <SQL template String> together with the placeholders for meta data. The SQL template for a foreign key statement will look like this (still sticking to the minimum syntax):

```
DECLARE        @ForeignKeyTemplate varchar(max) = '
ALTER TABLE    _ParentTableName_
ADD CONSTRAINT _fkName_
FOREIGN KEY  ( _ParentColumns_ )
REFERENCES     _ReferencedTableName_
             ( _ReferencedColumns_ )
';
```

Only one conversion to varchar(max) and the carriage returns are embedded in the string and not visible. Even though there is no color coding it's very recognizable foreign key statement.

The naming of the place holders and the meta data fields should match and be named wisely. I like a naming standard that conforms with the rules for regular identifiers, so you don't need to quote the data fields. Adding an underscore to the beginning and the end is a good enough naming convention. The placeholders stand out in the template and as a benefit the entire placeholder gets select when you double click in SSMS, this is true both within the SQL template as for the field values. It might seem as a small thing but it makes life a little bit easier.

Gathering of meta data

The meta data in this approach must be fetched from the same tables as in the ugly sample in the beginning of this post. But now the focus is on the individual parts that needs to go into the SQL template. The meta data is retrieved as separate fields and reusing the placeholders makes it easy read. My preference is to wrap it in a common table expression, this will separate the meta data retrieval from the replacing of the place holders.

```
with  cteFkeyMetaData
AS (
      select _ParentTableName_             = QUOTENAME(cs.name) + '.' + QUOTENAME(ptab.name)
           , _fkName_                      = QUOTENAME(fk.name)
           , _ParentColumns_               = STUFF(( SELECT     ',' + QUOTENAME(c.name)
                                                     FROM       sys.columns AS c
                                                     INNER JOIN sys.foreign_key_columns AS fkc
                                                             ON fkc.parent_column_id = c.column_id
                                                            AND fkc.parent_object_id = c.object_id
                                                     WHERE      fkc.constraint_object_id = fk.object_
                                                     ORDER BY   fkc.constraint_column_id
                                                     FOR XML PATH(''), TYPE).value('.', 'varchar(max)
           , _ReferencedTableName_          = QUOTENAME(rs.name) + '.' + QUOTENAME(rtab.name)
           , _ReferencedColumns_            = STUFF(( SELECT ',' + QUOTENAME(c.name)
                                                      FROM       sys.columns AS c
                                                      INNER JOIN sys.foreign_key_columns AS fkc
```

```
                                                    ON fkc.referenced_column_id = c.column_
                                                   AND fkc.referenced_object_id = c.object_
                                            WHERE      fkc.constraint_object_id = fk.object
                                            ORDER BY   fkc.constraint_column_id
                                            FOR XML PATH(''), TYPE).value('.', 'varchar(max
         , _delete_referential_action_desc_ = fk.delete_referential_action_desc COLLATE SQL_Latin1_Gen
         , _update_referential_action_desc_ = fk.update_referential_action_desc COLLATE SQL_Latin1_Gen
FROM       sys.foreign_keys fk
INNER JOIN sys.tables rtab
        ON fk.referenced_object_id = rtab.object_id
INNER JOIN sys.schemas rs
        ON rtab.[schema_id] = rs.schema_id
INNER JOIN sys.tables ptab
        ON fk.parent_object_id = ptab.object_id
INNER JOIN sys.schemas cs
        ON ptab.[schema_id] = cs.schema_id
 )
```

If you materialize your dynamic SQL creation in a SQL object, I prefer table valued functions, it's a good idea to return not only the complete SQL string but also the template and the replacement values. This will facilitate trouble shooting. When I started out working with dynamic SQL I used to wrap both construction of the SQL as string as well as the execution in one big stored procedure. It was both difficult to build and to debug.

Building the SQL string

What's left to do is to replace the place holders with the metadata fields. There will be a lot of nested REPLACE function, but if some effort is put into formatting and with field names matching the descriptive placeholders it easy to type as well as read.

```
SELECT ForeignKeyStmt = REPLACE(
                        REPLACE(
                        REPLACE(
                        REPLACE(
                        REPLACE(
                                  @ForeignKeyTemplate
                                , '_ParentTableName_' , _ParentTableName_)
                                , '_fkName_' , _fkName_)
                                , '_ParentColumns_' , _ParentColumns_)
                                , '_ReferencedTableName_' , _ReferencedTableName_)
                                , '_ReferencedColumns_' , _ReferencedColumns_)
     , ForeignKeyTemplate = @ForeignKeyTemplate
     , *
FROM   cteFkeyMetaData
```

To wrap up: Build the SQL String with a template and insert the meta data by using the REPLACE function. Let a query return the SQL string as well as the template and the meta data and your dynamic SQL life will be so much easier.

This entry was posted in Dynamic SQL and tagged Best practice. Bookmark the permalink.

Dynamic SQL and BIML
Proudly powered by WordPress.