The Curse and Blessings of Dynamic SQL

An SQL text by Erland Sommarskog, SQL Server MVP. Last revision: 2020-01-12.

1. Introduction

This is a text about dynamic SQL. Used in the right place, dynamic SQL is a tremendous asset, but dynamic SQL is also one of the most abused features in SQL Server. I frequently see posts in SQL forums that use dynamic SQL or ask for it, when there is no need to, or the desire to use dynamic SQL is due to an earlier mistake in the design. Quite often these posts are from inexperienced SQL users. It is important to understand that dynamic SQL is an advanced feature, and preferably you should not start using dynamic SQL until you have mastered to write static queries.

How much you should use dynamic SQL depends on your role. If you are an application developer, you should be restrictive with your use of dynamic SQL. Dynamic SQL certainly has its place in application code, but, as we shall see in this text, it also introduces complexity so there is all reason to be hold back. On the other hand, if you are a DBA, dynamic SQL is your best friend, because there are many DBA operations that can be automated with the help of dynamic SQL. (And at the same time as you use dynamic SQL all day long, you should attempt to prevent the devs from sticking dynamic SQL into the application code!)

Some readers may ask: what is dynamic SQL? Dynamic SQL is when you write SQL code into a string variable and then execute the contents of that variable. This can be done in client code or in a stored procedure. With a strict definition, static SQL can only occur in stored procedures and SQL scripts, since SQL in client code is always embedded into string literals in the client-side language. However, you could argue that a client that only has fixed SQL strings is using static SQL, and one of the aims of the article is to demonstrate how client code should be written to achieve this.

The first chapter after this introduction gives the basic commands to run dynamic SQL in a proper way and discusses traps you could encounter. This chapter also demonstrates how you should submit SQL queries from client code in a proper way. The next chapter is a very important one: it discusses SQL injection, a threat you must always protect yourself against when you work with dynamic SQL, no matter you are using dynamic SQL in stored procedures or you are writing client code. This is followed by a short chapter on the performance aspects of dynamic SQL and highlights a few rules you should follow. This chapter, too, is largely applicable both to stored procedures and client code.

From this point, however, the article leaves the client code aside and discusses only dynamic SQL in stored procedures and SQL scripts. The longest chapter in the article focuses on one of the big problems with dynamic SQL: if you don't have the discipline, code that generates dynamic SQL can easily become unreadable and difficult to maintain. This chapter is a style guide with tips to help you to write code that generates dynamic SQL in a better way. The penultimate chapter of this article discusses some good use cases for dynamic SQL. This chapter includes a section on how to run a dynamic pivot – something I see questions about daily in SQL forums. The last section covers situations where I often see dynamic SQL being abused, and which often are due to a bad design choice earlier in the process.

This article is intended for a broad range of SQL workers. Particularly the first chapters are written with the inexperienced SQL developer in mind, and seasoned DBAs and lead programmers may find the material overly basic. However, you may still be interested to read this material to get ammunition to keep your devs in check so that they don't abuse dynamic SQL. Experienced programmers may discover things in the style-guide chapter that you were not aware of or had not thought of. Conversely, the less experienced programmers may find that the latter part of this chapter a little difficult to digest, and you may prefer to drop out, and move on to the section on, say, dynamic pivot, if this is the problem you have at hand.

Table of Contents

1.1 Applicable SQL Versions and Demo Database

This article applies to all versions of SQL Server from SQL 2005 and up. There are no major differences between different SQL versions with regards to dynamic SQL since this release. There are a few smaller enhancements that have been added along the way, and I will call them out as we arrive at them.

However, this does not mean that all examples in the article will run on SQL 2005 as-is, but I have taken the liberty to embrace newer syntax where this helps to make the code shorter or more concise. These are small things, and you can easily modify the code if you want to run the examples on an older version of SQL Server. Some of these newer features appear only in a place or two, and for these I point out the differences as they appear. There are also a few features that I use over and over again, and which I list below and I generally don't mention further.

- **CREATE OR ALTER**. Throughout the article, I use CREATE OR ALTER PROCEDURE. This syntax was introduced in SQL 2016 SP1. On earlier versions you will have to use CREATE the first time and then change to ALTER if the same procedure is modified.
- **IIF**. IIF(*condition*, *X*, *Y*) is short for CASE WHEN *condition* THEN *X* ELSE *Y* END. IIF was introduced in SQL 2012, so with older versions you will need to replace IIF with the corresponding CASE expression.
- **INSERT VALUES for multiple rows**. This syntax was introduced in SQL 2008, so on SQL 2005 you will need to rewrite this with multiple INSERT VALUES statements.
- **DECLARE with initialisation**. This was added in SQL 2008, and with SQL 2005 you will need to split this into DECLARE + SET.

The demo database for this article is **NorthDynamic**, which you can create by running the script NorthDynamic.sql. The database is very small, less than 10 MB. The script runs on all versions of SQL Server from SQL 2005 and up. Not all examples build on this database; in those cases the article has CREATE TABLE statements where needed. I recommend that you work in **tempdb** for these examples.

NorthDynamic is a slightly modified version of Microsoft's old demo database **Northwind**, based on the fictive company Northwind Traders which flourished between 1996 and 1998; this explains why all orders are from this time frame. (If you are familiar with **Northwind** and wonder what the differences are: I have replaced deprecated data types so that the database can be installed with any collation. I have also made some modifications to schema and data for the benefit of some examples that I wanted to include. I gave the database a new name in case I want another variation of **Northwind** for another article.)

2. How to Run Dynamic SQL

In this chapter, we will look at how to run dynamic SQL in a civilised way. I like to point out there is nothing in the examples in this chapter that calls for dynamic SQL, and from that point of view they are bad examples, because you should not use dynamic SQL when there is no reason to. However, I opted to do it this way, because I wanted to focus on the essentials and keep the examples the simple. Actual use cases will come in the later chapters.

2.1 sp_executesql

sp_executesql is the system procedure that you invoke to run dynamic SQL. A way to describe this procedure is that it creates a nameless stored procedure which is saved to the plan cache (but not to disk) and then runs the procedure directly. On subsequent calls to this nameless procedure, the cached plan is reused. At this point however, we will not bother too much about the ramifications on the plan cache, but we will save that to the performance chapter. For the moment, just think of it as "creates a procedure and runs it directly".

And this is a very important thing: a batch of dynamic SQL is very similar to a stored procedure. About everything you can to do in a stored procedure, you can do in a batch of dynamic SQL. (There is one important difference with regards to permissions that I will cover in the next section, and a few more very marginal items that I will cover in this text.) And maybe the most important of all: you can use parameters. This is something which is very essential that you should make use of. The converse also applies: there is no syntax that is valid inside dynamic SQL, that is which is not valid in a stored procedure. All that is special is that the dynamic SQL is stored in a string and then executed.

sp_executesql has two fixed parameters: **@stmt** and **@params**. **@stmt** is the text of the SQL code to execute. The fact that the name is in singular is a little misleading: the SQL code does not have to be a single statement, but it can be a batch of any length with any number of statements. As you might guess, **@stmt** is a mandatory parameter. **@params** holds the parameter list for the code in **@stmt**, and the format of it is exactly like the format of the parameter list to a stored procedure. **@params** is not mandatory, but just like most of your stored procedures take parameters, most of your batches of dynamic SQL do. It is to the extent that every time you see a call to **sp_executesql** without parameters when reviewing code, there is all reason to check that the programmer is not doing the mistake of concatenating the parameter values into the string, which is a very bad thing to do. After **@params** follow the actual values to pass to the parameter list in **@params** in the same manner as you pass parameters to a regular stored procedure.

Here is an example script:

```
DECLARE @sql    nvarchar(MAX),
        @params nvarchar(4000)

SELECT @sql = N'SELECT @cnt = COUNT(*)
                FROM   dbo.Orders
                WHERE  OrderDate >= @date
                  AND  OrderDate < dateadd(MONTH, 1, @date)

                SELECT CompanyName
                FROM   dbo.Customers
                WHERE  CustomerID = @custid'

SELECT @params = N'@date    date,
                   @cnt     int OUTPUT,
                   @custid  nchar(5) = N''ALFKI'''

DECLARE @cnt  int,
        @when date = '19980201'

EXEC sp_executesql @sql, @params, @when, @cnt OUTPUT
SELECT @cnt AS [Orders in Feb 1998]

EXEC sp_executesql @sql, @params,
                   @date = '19970801', @cnt = @cnt OUTPUT, @custid = 'VINET'
SELECT @cnt AS [Orders in July 1997]
```

The script starts with declaring two variables which I later pass to **sp_executesql**. **@sql** is the variable I will pass to the **@stmt** parameter; I use the name **@sql** as this seems more natural to me. Observe that the datatype of the variable is **nvarchar(MAX)**. You should always use this data type for your batches of dynamic SQL, without any exception. You must use **nvarchar**, and cannot use **varchar**, of the simple reason that **sp_executesql** only accepts **nvarchar** and barfs if you attempt to use **varchar**. MAX, on the other hand, is not a requirement for **sp_executesql**, but it serves to protect your sanity. That is, if you use a limited type like **nvarchar(1000)**, there is the risk that the batch you compose exceeds the length you specify. This leads to silent truncation and errors that can drive you crazy as you are trying to understand what is going on.

For **@params**, I use **nvarchar(4000)**. As with **@stmt**, the data type must be **nvarchar**; **varchar** is not accepted. As for the length of 4000 that is mainly out of habit. You would have to have very many parameters to fill up that limit, but again, there is little reason to take the risk to make a guess at, say, 100, and then get errors because it was too low.

Next, I set up my SQL batch by assigning it to **@sql**, and since it is a static piece of code, there is little to remark on it. More interesting is the assignment to **@params** where I set up the parameter list. There are three parameters. The first is a "plain" parameter, whereas the second is an OUTPUT parameter and the third has a default value. Default values is nothing that is commonly used with dynamic SQL, but I included it here to emphasise that the parameter list is exactly like the parameter list to a stored procedure. On the other hand, using OUTPUT parameters with dynamic SQL is very common, since you often want to get scalar data back from your batch of dynamic SQL.

Before I invoke my dynamic SQL, I declare two local variables. Let's look at the call to **sp_executesql** again:

```
EXEC sp_executesql @sql, @params, @when, @cnt OUTPUT
```

The first parameter is **@sql**, that is, my SQL text, and the second is **@params**, that, is my parameter list. Next comes **@when**, my local variable that I initiated to 1998-02-01. This value is passed to the **@date** parameter in the SQL batch. Last comes **@cnt** which is to receive the value of the parameter **@cnt** in the dynamic SQL. Although they have the same name, they are really different entities; more about that later. Just as when I call a stored procedure, I need to specify OUTPUT also for the actual parameter. I don't pass a value for the parameter **@custid**, so the default for this parameter applies, and the second query returns the name *Alfreds Futterkiste* which is the full name for the customer with the ID ALFKI. (As for why customer IDs in **NorthDynamic** are codes rather than numeric IDs, I have no idea of what went on at Northwind Traders in the nineties, but it is kind of cute.)

There is a second call **sp_executesql** for the same SQL batch and parameters:

```
EXEC sp_executesql @sql, @params,
                   @date = '20180101', @cnt = @cnt OUTPUT, @custid = 'VINET'
```

As you can see, in this call I use named parameters rather than positional parameters, and for the input parameters **@date** and **@custid** I use literals for the actual parameters. This entirely in alignment with calls to regular stored procedures. Note particularly the value passed to the **@custid** parameter. This is a **varchar** literal (there is no N before it), while **@custid** is **nchar(5)**. That is, when we come to this part of the parameter list, we are free to mix **(var)char** and **n(var)char**. The restriction that we must pass **nvarchar** applies only to the fixed parameters **@stmt** and **@params**.

To drive the point that this is just like defining a stored procedure and running it at the same time, here is the same code logically, but with a "real " stored procedure:

```
CREATE OR ALTER PROCEDURE my_sp @date    date,
                                @cnt     int OUTPUT,
                                @custid  nchar(5) = N'ALFKI' AS

   SELECT @cnt = COUNT(*)
   FROM   dbo.Orders
   WHERE  OrderDate >= @date
     AND  OrderDate < dateadd(MONTH, 1, @date)

   SELECT CompanyName
   FROM   dbo.Customers
   WHERE  CustomerID = @custid
go
DECLARE @cnt  int,
        @when date = '19980201'

EXEC my_sp @when, @cnt OUTPUT
SELECT @cnt AS [Orders in Feb 1998]

EXEC my_sp @date = '19970701', @cnt = @cnt OUTPUT, @custid = 'VINET'
SELECT @cnt AS [Orders in July 1997]
```

Here is one more example to highlight one detail:

```
DECLARE @sql nvarchar(MAX) = 'SELECT CompanyName
                              FROM   dbo.Customers
                              WHERE  CustomerID = @custid'
EXEC sp_executesql @sql, N'@custid nchar(5)', 'VINET'
```

As you see, in this example I don't have any **@params** variable, but I define the parameter list directly in the call to **sp_executesql**. Observe the N preceding this parameter value; this is required to make it an **nvarchar** literal. I tend to use this pattern, if I have a short parameter list, but the longer the list is, the more likely that I will put the definition in a variable to make the code more readable. It is also possible to pass a literal value to the **@stmt** parameter, but since the SQL batch is almost often built from different parts, you will rarely do this practice.

2.2 Traps and Pitfalls

You have now learnt to use **sp_executesql** to run dynamic SQL. We will now look at some pitfalls that can surprise you, if your expectations are not in line with the actual functionality.

Scope of Variables

Consider this fairly nonsensical stored procedure:

```
CREATE OR ALTER PROCEDURE mishap_sp AS
   DECLARE @tbl TABLE (a int NOT NULL)
   DECLARE @a int = 99

   EXEC sp_executesql N'SELECT a FROM @tbl SELECT @a'
go
EXEC mishap_sp
```

When we run this, we get two error messages:

```
Msg 1087, Level 15, State 2, Line 1
Must declare the table variable "@tbl".
Msg 137, Level 15, State 2, Line 1
Must declare the scalar variable "@a".
```

If you have been paying attention, you already understand why we get this error: **sp_executesql** invokes a nameless stored procedure. That is, the code

```
SELECT a FROM @tbl SELECT @a
```

is not part of the procedure **mishap_sp** but of an inner (and nameless) stored procedure. As a consequence, the SELECT statement cannot access the variables in **mishap_sp**. In T-SQL, variables are only visible in the scope they are declared; never in inner scopes.

Note: *scope* is a word commonly used in programming to denote where variables (and other items) are visible. In T-SQL, a scope can be a stored procedure, trigger, function or simply an SQL script. A batch of dynamic SQL is also a scope.

If you want to pass variables from the surrounding procedure to the dynamic SQL, you must always pass them as parameters. For table variables, this requires that you have defined a table type, as in this example:

```
CREATE TYPE mytbltype AS TABLE (a int NOT NULL)
go
CREATE OR ALTER PROCEDURE hap_sp AS
   DECLARE @mytbl mytbltype,
           @a int = 99

   EXEC sp_executesql N'SELECT a FROM @tbl SELECT @a',
                      N'@tbl mytbltype READONLY, @a int', @mytbl, @a
go
EXEC hap_sp
```

Note: If you are still on SQL 2005: table-valued parameters were introduced in SQL 2008.

What Can Be Parameters?

Some people might try:

```
EXEC sp_executesql N'SELECT * FROM @tblname', N'@tblname sysname', 'Employees'
```

That is, they expect to be able to send in the table name as parameter. (As I will discuss in the section *Dynamic Table Names in Application Code* in last chapter, the urge to do so is often due to an earlier design mistake.) But the error message is:

```
Msg 1087, Level 16, State 1, Line 1
Must declare the table variable "@tblname".
```

There is nothing magic with dynamic SQL. You cannot put the name of a table in a variable and use it in a FROM clause. Nor can you put the name of a column in a variable and have it interpreted as such. And the same applies to almost all object names, with one exception: the name of a procedure you use in an EXEC statement. You will see examples of this as the article moves on.

Temp Tables and Dynamic SQL

In difference to the table-variable example earlier, this works:

```
CREATE OR ALTER PROCEDURE hap_sp AS
   CREATE TABLE #temp(b int NOT NULL)
   EXEC sp_executesql N'SELECT b FROM #temp'
go
EXEC hap_sp
```

This is because a temp table is visible in all inner scopes invoked by the module that created it.

On the other hand, this fails:

```
CREATE OR ALTER PROCEDURE mishap_sp AS
   EXEC sp_executesql N'SELECT * INTO #temp FROM dbo.Customers'
   SELECT * FROM #temp
go
EXEC mishap_sp
```

The error message is:

```
Msg 208, Level 16, State 0, Procedure mishap_sp, Line 3
Invalid object name '#temp'.
```

A local temp table is automatically dropped when the scope where it was created exits. That is, in this example **#temp** is dropped when the batch of dynamic SQL exits and therefore you cannot access it in the surrounding stored procedure. You need to create the table with CREATE TABLE before you invoke your SQL batch. ("But I don't know the schema, because it is dynamic!" some reader may object. If so, you have probably made a design mistake. We will return to that in the last chapter of this article.)

Dynamic SQL and Functions

You cannot use dynamic SQL in user-defined functions, full stop. The reason is simple: a function in SQL Server is not permitted to change database state, and obviously SQL Server cannot check beforehand what your dynamic SQL is up to. It also follows from the rule that you cannot call stored procedures in user-defined functions, and as we have learnt, a batch of dynamic SQL is a nameless stored procedure.

Permissions

This is one thing you need to be aware of, because here is something which is different from normal stored procedures. Consider this script where we create a test user which we impersonate to test permissions:

```
CREATE OR ALTER PROCEDURE mishap_sp AS
   SELECT COUNT(*) AS custcnt FROM dbo.Customers
   EXEC sp_executesql N'SELECT COUNT(*) AS empcnt FROM dbo.Employees'
go
CREATE USER testuser WITHOUT LOGIN
GRANT EXECUTE ON mishap_sp TO testuser
go
EXECUTE AS USER = 'testuser'
go
EXEC mishap_sp
go
REVERT
```

The output (with output set to text in SSMS):

```
custcnt
-----------
91

(1 row affected)

Msg 229, Level 14, State 5, Line 1
The SELECT permission was denied on the object 'Employees', database 'NorthDynamic', schema
'dbo'.
```

When **testuser** runs **mishap_sp**, the SELECT against **Customers** suceeds thanks to something known as *ownership chaining*. Because the procedure and the table have the same owner, SQL Server does not check if **testuser** has permission on the **Customers** table. However, the SELECT that is in the batch of dynamic SQL fails. This is because the batch of dynamic SQL is not considered to have an owner. Alternatively, the owner is considered to be the current user, which is **testuser**. Whichever, ownership chaining does not apply, and therefore SQL Server checks whether **testuser** has SELECT permission on **Employees** and since we never granted any permission to **testuser**, this fails.

Thus, if you consider using dynamic SQL, you need to talk with your DBA or whomever is in charge for the security in the database to verify that this is acceptable. Maybe the policy is that users should only have permission to run stored procedures, but they should not be granted any direct access to the tables. It is still possible to use dynamic SQL with such a policy in force, because you can bundle the permission with the stored procedure, if you sign it with a certificate and then grant a user created from that certificate the permissions needed. This is a technique that I describe in a lot more detail in my article *Packaging Permissions in Stored Procedures*. While this is a perfectly possible and valid solution, it does increase the complexity of your system a little bit, and thus raises the hurdle for using dynamic SQL.

You Are Hiding Your References

If you want to know where a certain table is referenced, SQL Server offers a couple of ways to determine this. There are the system procedure **sp_depends** and the catalog views **sys.sql_dependencies** and **sys.sql_expression_dependencies**. You can also use *View Dependencies* from the context menu of a table in the Object Explorer in SSMS; under the hood SSMS uses **sys.sql_expression_dependencies**. However, none of these will show you any references made in dynamic SQL, since when SQL Server parses a stored procedure and saves the dependencies, the dynamic SQL in just in a string literal which SQL Server has no reason to care about at that point.

Note: I offer a simple utility SearchCode which reads all SQL code in a database into a full-text indexed table which permits you to search the code using full-text operators such as CONTAINS and thereby you can find references also in dynamic SQL.

Chintak Chhapia pointed out a problem of similar nature: when you are looking into to upgrade your SQL Server instance to a newer version, you may want to use the Upgrade Advisor to find if there are any compatibility issues in your code or whether you are using any deprecated features. However, the Upgrade Advisor is entirely blind for what you do in dynamic SQL.

The RETURN statement

Beside the differences with permissions, there are a few more small differences between dynamic SQL and stored procedures. None of them are very significant, but here is one that Jonathan Van Houtte ran into. He wanted to use a RETURN statement with a specific return value in his dynamic SQL, but this is only permitted in regular stored procedures. That is, this fails:

```
EXEC sp_executesql N'IF @x = 0 RETURN 122', N'@x int', 0
```

The error message is:

```
Msg 178, Level 15, State 1, Line 1
A RETURN statement with a return value cannot be used in this context.
```

You can still use RETURN without a return value in dynamic SQL. This is legal:

```
EXEC sp_executesql N'IF @x = 0 RETURN', N'@x int', 0
```

2.3 EXEC()

EXEC() is an alternate way to run dynamic SQL. It exists of historic reasons, see more the note at the end of this section. While there is never any compelling reason to ever use this form, there are plenty of examples of it out the wild, so it would be wrong to be silent on it.

EXEC() does not support parameters, so all values have to be inlined. Here is an example, similar to the one we looked at previously:

```
DECLARE @sql     varchar(MAX),
        @when    date = '1998-02-01',
        @custid  nchar(5) = N'ALFKI',
        @empid   int = 3

SELECT @sql =
   'SELECT COUNT(*)
    FROM   dbo.Orders
    WHERE  OrderDate >= ''' + convert(char(8), @when, 112) + '''
      AND  OrderDate < ''' + convert(char(8), dateadd(MONTH, 1, @when), 112) + '''
      AND  EmployeeID = ' + convert(varchar(10), @empid) + '

    SELECT CompanyName
    FROM   dbo.Customers
    WHERE  CustomerID = N''' + @custid + ''''

PRINT @sql
EXEC(@sql)
```

If you look at this code and compare it with the original example with **sp_executesql**, what do you think? Don't you get the impression that this looks a lot more cumbersome? The logic of the queries is more difficult to follow when the SQL is broken up by all those single quotes, plusses and calls to **convert**. The reason for this cascade of single quotes is that to include a single quote in a string literal, you need to double it, and typically the single quotes appear at the beginning or the end of a string fragment so the double single quote inside the string is followed by the single quote that terminates the string. (If that sentence got your head spinning, you sort of got my point. :-)

When it comes to the **convert**, date and time values constitute a special challenge, since you need to pick a good format code, so that the constructed date string is not misinterpreted. 112 is a good code for dates only, 126 for values with both date and time.

You may note here that **@sql** is declared as **varchar(MAX)**; in difference to **sp_executesql**, EXEC() accepts both **varchar** and **nvarchar**.

As you can see, I have added a diagnostic PRINT, so that you can see the actual SQL generated. This is the output:

```
SELECT COUNT(*)
    FROM   dbo.Orders
    WHERE  OrderDate >= '19980201'
      AND  OrderDate < '19980301'
      AND  EmployeeID = 3

    SELECT CompanyName
    FROM   dbo.Customers
    WHERE  CustomerID = N'ALFKI'
```

This is something I will return to in the style guide, but it is worth making the point already now: when you work with dynamic SQL and build a query string, you always need a diagnostic PRINT, so that you can see what you have generated in case you get an error message or an unexpected result.

The input to EXEC() does not have to be a single variable, but it can be a concatenation of simple terms, that is, variables and literals, as in this example::

```
DECLARE @custid  nchar(5) = N'ALFKI'

EXEC('SELECT CompanyName
      FROM   dbo.Customers
      WHERE  CustomerID = N''' + @custid + '''')
```

Functions calls are not permitted, so this fails with a syntax error:

```
DECLARE @when    date = '1998-02-01',
        @empid   int = 3

 EXEC(
   'SELECT COUNT(*)
    FROM   dbo.Orders
    WHERE  OrderDate >= ''' + convert(char(8), @when, 112) + '''
      AND  OrderDate < ''' + convert(char(8), dateadd(MONTH, 1, @when), 112) + '''
      AND  EmployeeID = ' + convert(varchar(10), @empid) + )
```

In any case, there is little reason to use this pattern. If things go wrong, there is no way to splice in a diagnostic PRINT, so it is much better to store the SQL string in an **@sql** variable.

Overall, there is little reason to use EXEC(), when **sp_executesql** permits you to use parameters and thereby making the code a lot more readable. And, as we shall see later in this article, readability and complexity is only one reason to stay away from string concatenation. So I could make it short and say *Don't use EXEC(). Ever.* But that would not be honest, because I tend to use EXEC() myself when the SQL string takes no parameters. That is, rather than saying

```
EXEC sp_executesql @sql
```

I most often write the shorter:

```
EXEC(@sql)
```

You will see this more than once in this article. But you could certainly argue that this is just a bad habit of mine. If you decide to always use **sp_executesql**, no matter whether your dynamic SQL takes parameters or not, you would not being do something wrong.

Before I close this section, I should add that there is actually one thing you can do with EXEC() that you cannot do with **sp_executesql**: you can impersonate user or a login when your run your SQL batch. So instead of the impersonation we did above when we tested permissions for **testuser**, we could have done:

```
EXEC ('EXEC mishap_sp') AS USER = 'testuser'
```

This has the advantage that you know for sure that you will revert from the impersonated context, even if there is an execution error. But personally, I don't think this outweighs the increased complexity that dynamic SQL incurs. In any case, this has to be considered as an advanced feature.

Note: EXEC() may come to use on very old versions of SQL Server. **sp_executesql** was introduced in SQL 7, so if you find yourself on SQL 6.x, EXEC() is your sole option. You may also have to use EXEC() on SQL 7 and SQL 2000, if your dynamic SQL can exceed 4000 characters, since **nvarchar(MAX)** is not available on these versions. EXEC() permits you to work around this, since you can say EXEC(**@sql1** + **@sql2** + **@sql3**) and it accepts that the resulting string exceeds 8000 bytes.

2.4 EXEC() AT linked server

This is a special form of EXEC() which permits you to run a batch of commands on a linked server. This form of EXEC() permits you to use parameters, if in a different way from **sp_executesql**.

To play with this, you can set up a linked server that goes back to your current instance like this:

```
EXEC sp_addlinkedserver LOOPBACK, '', 'SQLNCLI', @datasrc = @@servername
EXEC sp_serveroption LOOPBACK, 'rpc out', 'true'
```

You can replace **@@servername** with the name of another instance you have access to if you like. Just make sure that you have **NorthDynamic** on that instance as well.

Here is how you could run the batch of dynamic SQL above:

```
DECLARE @sql     varchar(MAX),
        @when    date = '19980201',
        @custid  nchar(5) = 'VINET',
        @cnt     int

SELECT @sql = 'SELECT ? = COUNT(*)
               FROM   NorthDynamic.dbo.Orders
               WHERE  OrderDate >= ?
```

```
                 AND  OrderDate < dateadd(MONTH, 1, ?)

               SELECT CompanyName
               FROM   NorthDynamic.dbo.Customers
               WHERE  CustomerID = ?'
PRINT @sql
EXEC(@sql, @cnt OUTPUT, @when, @when, @custid) AT LOOPBACK
SELECT @cnt AS [Orders in Feb 1998]
```

In difference to **sp_executesql**, the parameters do not have names, but instead the question mark serves as a parameter holder. SQL Server will pass the parameters given to EXEC() AT in the order the question marks appear. That is, **@cnt** goes to the first question mark, and as you see we can use the OUTPUT keyword here as well. Next **@when** comes twice in the parameter list, simply because it used twice in the SQL code.

There is a good reason why EXEC() AT does not use parameter names like **sp_executesql**: the linked server may be something else than SQL Server where names starting with @ would make little sense. The ? parameter marker on the other hand is a standard thing in ODBC and OLE DB, and the OLE DB provider for the remote data source will change the ? into whatever makes sense on the other data source.

For simplicity, I used EXEC() AT with a linked server that is another SQL Server instance. However, in this case it is a lot easier to use **sp_executesql** with four-part notation as in this example:

```
DECLARE @sql    nvarchar(MAX),
        @params nvarchar(4000)

SELECT @sql = N'SELECT @cnt = COUNT(*)
                FROM   dbo.Orders
                WHERE  OrderDate >= @date
                  AND  OrderDate < dateadd(MONTH, 1, @date)

                SELECT CompanyName
                FROM   dbo.Customers
                WHERE  CustomerID = @custid'

SELECT @params = N'@date    date,
                   @cnt     int OUTPUT,
                   @custid  nchar(5) = N''ALFKI'''

DECLARE @cnt  int,
        @when date = '19980201'

EXEC LOOPBACK.NorthDynamic.sys.sp_executesql @sql, @params, @when, @cnt OUTPUT
SELECT @cnt AS [Orders in Feb 1998]
```

You may note that in the example with EXEC() AT, I included the database name in the SQL string, but I could avoid it in the version with **sp_executesql**. When you invoke **sp_executesql** (as any other system procedure) with a given database name, **sp_executesql** will run in the context of that database. Thus, there is all reason to use this pattern when you want to run a parameterised SQL statement on a remote SQL Server instance. But when your linked server is not running SQL Server, EXEC() AT is definitely your friend.

Keep in mind that there are some restrictions in what data types you can use on linked servers. For instance, you cannot use the **xml** data type. You can pass LOB types (e.g. **nvarchar(MAX)**), but only as input parameters; you cannot receive them as OUTPUT parameters.

2.5 Running Dynamic SQL From Client Code

The difference between stored procedures and client code in this regard is that in stored procedures you only use dynamic SQL occasionally, but in client code you could say that you do it all the time since the SQL code is always inside a string literal in the client-side language. However, that does not mean that you always will have to use string concatenation to build the string. Au contraire, this is an exceptional case. As long as you are only working with plain-vanilla queries against specific tables, your queries should be contained in a single string literal that holds the full query and where the input values from the user are passed as parameters. Or put in a different way: no matter whether you are doing dynamic SQL code in a stored procedure or in client code, anything that can be a variable as permitted by the SQL syntax, should be passed as parameter and not be inlined into the query string by means of concatenation. In this section we will learn how to do this in .NET, and we will also start looking at the many problems with inlining parameter values.

Note: If you are working in a different environment than .NET, you should still read this section. The pattern I introduce here apply to all client environments for SQL Server, although the name of the objects and the methods are different. And while .NET supports named parameters in the **@param** style, some environments only support parameters markers, typically ?, as we saw examples of in the previous section.

I should also point out that this section aims at client code where you actually write SQL code. If you are using an ORM such Entity Framework you may be using a language like Linq that generates the SQL code. For reasons irrelevant to this article, I am not a friend of ORMs, but at least they keep users away from writing bad dynamic SQL, and I will not cover Linq and similar in this article.

For this section, we will not use the tables in **NorthDynamic**, but instead work with this table:

```
CREATE TABLE typetbl (ident   int           NOT NULL IDENTITY,
                      intcol  int           NOT NULL,
                      deccol  decimal(8,2)  NOT NULL,
                      fltcol  float         NOT NULL,
                      strcol  nvarchar(25)  NOT NULL,
                      dtcol   datetime      NOT NULL,
                      CONSTRAINT pk_datatyptbl PRIMARY KEY (ident)
)
```

While you could create it in **NorthDynamic**, I recommend that you create the table in **tempdb**, and this is what I will assume in the text that follows.

Say that we want to insert a row into this table and we have the values in variables **intval**, **decval** etc and we want to get back the value for **ident** for the inserted row. Here is a method to do this:

```
public static void InsertTbl () {

   using(SqlConnection cn = new SqlConnection(strConn)) {
   using(SqlCommand cmd = new SqlCommand()) {
      cmd.CommandType = CommandType.Text;

      cmd.CommandText =
           @"INSERT dbo.typetbl(intcol, strcol, fltcol, deccol, dtcol)
                 VALUES(@intval, @strval, @fltval, @decval, @dtval)
             SELECT @ident = scope_identity()";

      cmd.Parameters.Add("@intval", SqlDbType.Int).Value = intval;
      cmd.Parameters.Add("@strval", SqlDbType.NVarChar, 25).Value = strval;
      cmd.Parameters.Add("@fltval", SqlDbType.Float).Value = fltval;
      cmd.Parameters.Add("@decval", SqlDbType.Decimal).Value = decval;
      cmd.Parameters["@decval"].Precision = 8;
      cmd.Parameters["@decval"].Scale = 2;
      cmd.Parameters.Add("@dtval", SqlDbType.DateTime).Value = dtval;
      cmd.Parameters.Add("@ident", SqlDbType.Int);
      cmd.Parameters["@ident"].Direction = ParameterDirection.Output;

      cmd.Connection = cn;
      cn.Open();

      cmd.ExecuteNonQuery();

      int identval = Convert.ToInt32(cmd.Parameters["@ident"].Value);
      Console.WriteLine("The inserted row has id " + identval.ToString());
   }}
}
```

We first create connection and command objects (inside `using` clauses to make sure that resources are released as soon they go out of scope) and set the command type to be text. Next, in adherence with what I said in the beginning of this section, we define the command text as a single static string which includes a couple of T-SQL variables. Or more precisely, they are *parameters*.

In the suite of statements that follows, we define these parameters with the **Add** method of the **SqlParameterCollection** class. There are a couple of overloads of this method, of which the two most commonly used appear in this example. The first parameter is the name of the parameter. (It appears that the @ can be left out, but I recommend that you always included it.) The second parameter is a value from the enumeration **SqlDbType** (which is in the **System.Data** namespace). This enumeration basically has one entry for each data type in SQL Server, with the name being the same as the SQL Server data type, but with initial uppercase and some middle uppercase as well to spice it up. There are a few deviations with some lesser used types that I will not cover here. For a complete list, see the.NET Framework reference. For types that go with a length, that is, string and binary types, we use an overload that takes a third parameter which is the length. (For LOB types such as **nvarchar(MAX)**, you specify -1 for the length.) The example includes a decimal value. There is no overload to define a **decimal** parameter with precision and scale, so we need set these properties separately as seen in the code sample above.

The **Add** method returns the newly created **SqlParameter** object which permits us to set the **Value** property directly, and thus for all but **@decval** we can define a parameter in a single line of code. I did not include this in the example, but to explicitly pass a NULL value, set the **Value** property to **System.DBNull.Value**.

For **@ident** there is no value to set, since this is an output parameter, which we indicate by setting the **Direction** property. (This property defaults to **ParameterDirection.Input**.)

Once the parameters have been defined, we open the connection, run the SQL batch and then we retrieve the generated IDENTITY value from the output parameter and print it out.

At this point you may want to try the code. The file clientcode.cs includes **InsertTbl** as well as a method **BadInsert** that we will look at later. There is also a **Main** that parses the command line and has a simple exception handler. In the top of the file you see this:

```
//  Connection string, change server and database as needed!
private static string
    strConn = @"Integrated Security=SSPI;" +
              @"Data Source=(local);Initial Catalog=tempdb;";

// Constants for the demo.
private static int      intval = 123456;
private static string   strval = "alpha (α) and beta (β)";
private static double   fltval = 17.0/3.0;
private static decimal  decval = 456.76M;
private static DateTime dtval  = DateTime.Now;
```

Save and compile the program. (See here if you need assistance on compilation.) Open a command prompt to move to the folder where you have the executable and run it as:

```
clientcode
```

You should see an output like this:

```
The inserted row has id 1
```

You can also run a SELECT against **typetbl** to see that the values were inserted.

Let's now investigate what happens in terms of T-SQL. You can use Profiler to capture the command that is sent to SQL Server. (You can also use the XEvent Profiler in recent versions of SSMS). You will see something like this: (I have reformatted the output for legibility):

```
declare @p8 int
set @p8=2
exec sp_executesql N'INSERT dbo.typetbl(intcol, strcol, fltcol, deccol, dtcol)
                 VALUES(@intval, @strval, @fltval, @decval, @dtval)
             SELECT @ident = scope_identity()',
             N'@intval int, @strval nvarchar(25), @fltval float,
               @decval decimal(8,2), @dtval datetime, @ident int output',
       @intval=123456, @strval=N'alpha (α) and beta (β)', @fltval=5.666666666666667,
       @decval=456.76, @dtval='2018-09-22 00:07:35.193', @ident=@p8 output
```

```
select @p8
```

So that was the heart of the matter! The @ parameters are not just an abstraction in .NET, but the batch is passed to **sp_executesql** exactly as it was entered and all the parameters are part of the parameter list in **@params**.

The full story is a little different, though. .NET does not actually compose the code above, but what you see is a textual *representation* of what largely is a binary stream of data. What appears in the trace is an RPC event (RPC = Remote Procedure Call). That is, .NET tells SQL Server to run a stored procedure, this time **sp_executesql**, with so and so many parameters, and then it sends the parameters in *binary* form. If were you to eavesdrop on the wire with Wireshark or similar, you would see the INSERT statement, but not the surrounding quotes. If you were to look at the bytes preceding the INSERT keyword, you would find the length of the string. The same goes for the parameters **@params** and **@strval**. You would not be able to discern the values of **@intval**, **@fltval**, **@decval** and **@dtval**, unless you are familiar with the binary format.

Note: The variable **@p8** in the Trace output is a very funny way that Profiler uses to indicate the return value of output parameters. In reality, **@p8** does not exist at all. And particularly, the value 2 is never passed to the **@ident** parameter.

The reader may find this to be a little too much of nitty-gritty details to really catch your attention, so let me summarise what are the important points so far:

- When you define one or more parameters for your SQL batch, this results in a call to **sp_executesql**.
- The parameter values are passed in a binary format without any SQL syntax around them. (Why this matters, will be apparent later.)

And most of all:

- When you pass values to an SQL batch from your client code, be that .NET or anything else, you should always use parameterised statements.

The last point should really be a superfluous one to make, because what you have seen is the norm that any professional software developer should follow. But unfortunately, I see too far many examples in user forums and elsewhere which indicate that this is not always the case. Therefore, we need to look at what far too many people do, and why you should not do that. In clientcode.cs there is also a method **BadInsert** which looks like this:

```
public static void BadInsert () {

   using(SqlConnection cn = new SqlConnection(strConn)) {
   using(SqlCommand cmd = new SqlCommand()) {
      cmd.CommandType = CommandType.Text;
      cmd.CommandText =
            @"INSERT typetbl (intcol, strcol, fltcol, deccol, dtcol)
              VALUES(" + intval.ToString() + @",
                    '" + strval + @"',
                     " + fltval.ToString() + @",
                     " + decval.ToString() + @",
                    '" + dtval.ToString() + @"')
              SELECT scope_identity()";

      Console.WriteLine(cmd.CommandText);

      cmd.Connection = cn;
      cn.Open();

      Object res = cmd.ExecuteScalar();

      int identval = Convert.ToInt32(res);
      Console.WriteLine("The inserted row has id " + identval.ToString());
   }}
}
```

At first glance it may seem shorter, and some readers may even find it simpler than **InsertTbl**, as there is no call to extra methods. "We just build the query string, let's stick to the KISS principle and skip the fancy stuff". (KISS = *Keep it Stupid and Simple.*) But, as we shall see, this is only stupid – it is not simple at all. The fact that I felt compelled to add a diagnostic write of the resulting command text may be a first indication of this.

If you are hot on it, you can try it immediately by running this in the command-line window:

```
clientcode BAD
```

This will invoke **BadInsert**. However, I should warn you that for a completely successful result all the points below must be true:

1. In your regional settings in Windows (or locale on Linux), the decimal separator must be a period and not a comma.
2. The date format in your regional settings must match the date-format setting in SQL Server.
3. Your database must have a Greek collation.

When I ran the program in on my computer, I got this output:

```
INSERT typetbl (intcol, strcol, fltcol, deccol, dtcol)
                 VALUES(123456,
                       'alpha (a) and beta (ß)',
                        5,66666666666667,
                        456,76,
                       '2018-09-22 23:25:57')
                 SELECT scope_identity()
EXCEPTION THROWN: There are fewer columns in the INSERT statement than values specified in
the VALUES clause.
The number of values in the VALUES clause must match the number of columns specified in the
INSERT statement.
```

As you see, an exception was thrown and no data was inserted. Furthermore, the error message seems mysterious at first. When we go back to the .NET code, and count columns and values, we can find no mismatch. But if you look more closely at the generated command you can find commas in the values for **deccol** and **fltcol**, which messes up the T-SQL syntax. These commas appeared because **ToString** respects the regional settings and my I have my regional settings in Windows set to Swedish, where the decimal delimiter is a comma and not a period. There is obviously a simple fix: use something more sophisticated than **ToString** that permits you to control the format. However, imagine a naïve programmer living in, say, Australia, where period is the decimal delimiter, who writes the above. The code works, passes all tests and is shipped. Then at some point it reaches a part of the world, for instance continental Europe, where comma is the decimal delimiter. And European users find that the software crashes as soon as they start it. This is not really aimed to give credibility in the software vendor.

To be able to run the program, I changed my regional settings to English (Australia). Now I got a different error:

```
INSERT typetbl (intcol, strcol, fltcol, deccol, dtcol)
                 VALUES(123456,
                       'alpha (a) and beta (ß)',
                        5.66666666666667,
                        456.76,
                       '22/09/2018 11:33:51 PM')
                 SELECT scope_identity()
EXCEPTION THROWN: The conversion of a varchar data type to a datetime data type resulted in
an out-of-range value.
The statement has been terminated.
```

This time, the decimal and float values are good, but instead the value for **dtcol** causes a problem. My language setting in SQL Server is us_english and the date-format setting is **mdy**. That is, SQL Server expects the value to be month/day/year. When I ran this on the 22nd of the month, I got the error. For the first twelve days of the month, the value would have been inserted – but for eleven of those days it would have been inserted wrongly! Again, this is something can be addressed by formatting: the date should be in a format that is safe in SQL Server, for instance YYYY-MM-DDThh:mm:ss.fff. (The T in this format stands for itself.) But a naïve developer in the US will never notice the problem, because he has the same setting in Windows and SQL Server. (And it is not any better with a naïve Swedish developer for that matter. The Swedish format YYYY-MM-DD is correctly interpreted with the **mdy** setting.) In this case, the software may even work at sites in countries like Germany or Australia where dates are written as day/month/year, if they have the date format set to **dmy** in SQL Server. But sooner or later there will be a mismatch, and there will be errors. Or even worse, incorrect dates.

Note: for a more detailed discussion on date formats, see the article *The ultimate guide to the datetime datatypes* by SQL Server MVP Tibor Karaszi.

I made one more attempt to run the program. This time I set my regional settings to English (United States). I got this output:

```
INSERT typetbl (intcol, strcol, fltcol, deccol, dtcol)
                 VALUES(123456,
                       'alpha (a) and beta (ß)',
                        5.66666666666667,
                        456.76,
                       '9/22/2018 11:50:47 PM')
                 SELECT scope_identity()
The inserted row has id 4
```

Success! The row was inserted. However, when I looked at the data, I noticed a snag. In **strcol** I saw this:

```
alpha (a) and beta (ß)
```

If you look closely, you can see that the Greek alpha has been replaced by a regular lowercase Latin lowercase *a*, and instead of a lowercase beta, there is a German "scharfes s". This is due to that there is no N before the string literal in the SQL statement. Thus, this is a **varchar** literal, and for **varchar**, the character repertoire is defined by the code page for the collation. My collation for the test was a Finnish_Swedish collation, which uses code page 1252 for **varchar**, and this code page does not include alpha and beta. Therefore, SQL Server replaced them by fallback characters it deemed suitable.

Note: the observant reader may notice that the same replacement occurred also in the output in the command-line window when running the program. This is not because of the missing N, but because the command-line environment has no Unicode support at all.

There are more problems with **BadInsert**. The **clientcode** program permits you to override the value for **strval** on the command line. Run this:

```
clientcode BAD Brian O'Brien
```

I had changed back my regional settings to Swedish when I ran this, but it does not really matter.

```
INSERT typetbl (intcol, strcol, fltcol, deccol, dtcol)
                 VALUES(123456,
                       'Brian O'Brien',
                        5,66666666666667,
                        456,76,
                       '2018-09-23 00:07:36')
                 SELECT scope_identity()
EXCEPTION THROWN: Incorrect syntax near 'Brien'.
Unclosed quotation mark after the character string ')
SELECT scope_identity()'.
```

Because of the single quote in Brian's name, this time I got a syntax error. Now look at this:

```
clientcode GOOD Brian O'Brien
```

GOOD invokes **InsertTbl** and the row is inserted without any pain.

Yes, we can fix **BadInsert** so that it doubles the quotes in **strval**. But is that going to make the SQL code easier to read?

And that brings us to the gist of this exercise. **InsertTbl** shows a very structured way of writing data-access code. You define the SQL statement in a single string, which is free from any disruption of .NET syntax, save for the surrounding double quotes. Then it is followed by a definition of the parameters in a simple but structured way. In **BadInsert**, the SQL statement is mixed with .NET syntax, which means that the more variables there are, the more difficult it is to see the SQL forest for all the .NET trees.

With **BadInsert** you can get a clash between the regional settings in the client and the interpretation in SQL Server which has its own settings and rules, different from .NET and Windows. On the other hand, with **InsertTbl**, all values are interpreted consistently in the client program according to the regional settings on the user's computer and they are transformed to an unambiguous binary format that SQL Server always interprets in the same way.

When it comes to the final mistake, leaving out the N, this is an error you could commit with **InsertTbl** as well, as you could mistakenly use **SqlDbType.VarChar** instead of **SqlDbType.NVarChar** which would lead to the same mutilation of the Greek characters. Still, the .NET interface sort of encourages you to be exact with the data types.

The observant reader may have noted that there is one more difference between the two methods. In **InsertTbl**, I use an output parameter to get the id value of the inserted row, whereas in **BadInsert** I return the id with a SELECT statement and I use **ExecuteScalar** to run the batch and to get the value that way. Both of these approaches are perfectly valid, and I wanted to show examples with both. Which one you use is a matter of taste and what fits in best with the moment. (Obviously, if you have more than one return value, **ExecuteScalar** is not an option.)

Some readers may feel that having to specify the exact data type and details like length of strings and precision/scale for **decimal** is a bit of a burden. Surely .NET which is aimed at increasing developer productivity, could do better? Indeed. If you don't specify the length for a string or binary parameter, .NET will use the length of the value you supply, and the similar is true for decimal values. Furthermore, the **SqlParameterCollection** class provides the method **AddWithValue** which infers the type from the value you pass in. NEVER USE ANY OF THIS, PERIOD! You see, while this a boost to developer productivity, it is a setback for the productivity of DBA who is likely to be the one who has to figure out why the database access is slow. Why it is so, is something I will return to in the performance chapter. But if you want to know more details here and now, you can read the blog post *AddWithValue is Evil* from SQL Server MVP Dan Guzman.

You may argue that mirroring the length of strings in the .NET application causes a maintenance problem and you don't want to change the code if the column is changed from **nvarchar(25)** to **nvarchar(50)**. If that is a concern, you can use 100 as a standard value for all "short" columns and 4000 for all longer **nvarchar** and 8000 for long **varchar**. (For MAX parameters, you should use -1 as noted above.) There are still situations where this can affect performance, but this something which a lot more subtle, and it is nothing I will discuss in this article. And the impact is nowhere as far-reaching as when you don't specify any length at all.

In this section, I only looked at .NET, but whichever environment you are using, building an SQL string by concatenating user input is totally unacceptable, if you are writing code in a professional context as an employee or a consultant. You must always use parameters where the SQL syntax permits. This may seem like a very harsh thing to say, but in this section we only looked at it from the point of view of developer ease. Writing and maintaining code with fixed parameterised SQL strings is so much easier than inlining parameter values. We will look at an even more important reason in the next chapter from which it will be clear why I use such strong words. Note that we will keep working with **typetbl** and the **clientcode** program, so don't drop them now.

- -

3. Beware of SQL Injection

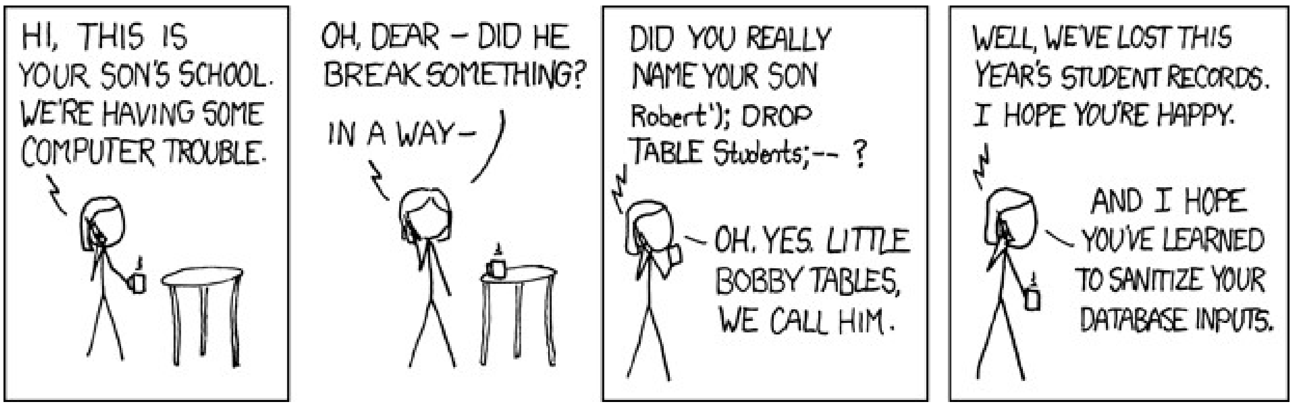
When you work with dynamic SQL you need to be aware of SQL injection. Well, "aware" is a bit of understatement. Every time you build code dynamically from some input, you should ask yourself these questions:

1. Can the input be crafted so that the generated SQL code performs something else than intended?
2. Can this be utilised by a malicious user?
3. Can I protect myself against it?

In this chapter, we will look at how the first two things can happen and what your means of protection are. The first section discusses SQL injection in client code, but this does not mean that you can skip this section if you only do T-SQL programming, because the lessons in this chapter are of a general nature and applies to everyone. The remaining sections in this chapter covers aspects specific to T-SQL.

3.1 SQL Injection in Client Code

What is SQL injection? SQL injection is when a user enters SQL syntax in an input field or similar and causes the application to perform something else than intended. Here is a classic example, *Exploits of a Mom*, from the comic strip **xkcd**.


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

For the reader who have never heard of the concept of SQL injection, this may not register right away, but let's try it with our **clientcode** program. Let's first create a **Students** table:

```
CREATE TABLE Students (a int NOT NULL)
```

(Since we are going to try to drop it, the columns do not really matter.)

To see if xkcd might be pulling our legs, we try the name as given in the strip:

```
clientcode BAD Robert'); DROP TABLE Students; --
```

No, this did not work out:

```
INSERT typetbl (intcol, strcol, fltcol, deccol, dtcol)
                VALUES(123456,
                      'Robert'); DROP TABLE Students; --',
                       5,66666666666667,
                       456,76,
                      '2018-09-23 21:47:00')
                SELECT scope_identity()
EXCEPTION THROWN: Incorrect syntax near '5'
```

But if you look more closely at this, you may see that xkcd is on to something. That single quote after Robert seems to fit into the syntax. The single quote that originally was in **BadInsert** is still there – but it is hidden behind the two hyphens that start a comment. As it happens, I introduced newlines in the generated statement to make it easier to read, and therefore the number on the next line causes a syntax error. Many developers would not bother about adding newlines, and with the statement on a single line, the generated command would have been syntactically correct as such. It would still have failed due to the mismatch of number of columns between the INSERT and VALUES list.

None of those are obstacles that cannot be overcome. Next try this:

```
clientcode BAD Robert', 1, 1, 1); DROP TABLE Students; SELECT 1 WHERE 1 IN ('1
```

It may look like a big piece of nonsense, but look at the output:

```
INSERT typetbl (intcol, strcol, fltcol, deccol, dtcol)
                 VALUES(123456,
                       'Robert', 1, 1, 1); DROP TABLE Students; SELECT 1 WHERE 1 IN ('1',
                        5,66666666666667,
                        456,76,
                       '2018-09-23 21:51:48')
                 SELECT scope_identity()
The inserted row has id 1
```

The batch completed without errors and if you run `SELECT * FROM Students` in a query window, you find that the table is not there.

To understand how **BadInsert** was exploited, let's look at the techniques used. To get the original INSERT statement to work, we added the digit 1 as many times as needed to make VALUES to match INSERT. And we were happy to see that 1 was accepted for all values (there could have been conversion errors or constraint violations). To get the rest of the syntax to work, we had to incorporate that list of values in a new statement, and a SELECT with IN seemed like the easiest pick. We don't bother about a table in the SELECT, because there is nothing that says that a column or a variable must come before IN – it can just as well be a literal. Finally, we had to consider the quote that is intended to close the value in **strval** and the comma that follows. As you can see, this was achieved by the final `'1`.

Just as a contrast: recreate the **Students** table and run the same again, but replacing BAD with GOOD. This also completes successfully, but when you check the **Students** table, it is still there. If you look in **typetbl**, you find the first 25 characters of the injection string, and in **fltcol**, **deccol** and **dtcol** you find the constant values from the clientcode.cs file, that is, the intended values. On the other hand, in the row inserted in the injection attack we see something else. (The result of all those 1s.) You may also note that the **ident** column for the row added with **BadInsert** has a different value than was reported in the output above. This is because **ExecuteScalar** picked up that 1 from the extra SELECT we injected.

It was easy for us to compose the injection string, because we had access to the source code. You may argue that it would be a lot more difficult for an intruder who don't see code to find out how the injection attack should be crafted, but that is not really the case. Imagine that that **BadInsert** is actually inside a web application which is open to everyone on the Internet and **strval** gets its value from an URL. Or for that matter, a cookie. The evil user may first attempt a simple type of exploit in the xkcd strip. If the user is lucky, the web server displays the error message from SQL Server, and in such case it's a relatively simple task to figure it out. That would be bad practice in itself; best practice is to configure the web server to show a generic error message and only write the actual error message to a local log file. But that is still not a roadblock, only a small speedbump that may stop someone who is trying to find an injection hole manually. In real life, intruders use automated tools to find SQL injection holes. A site which is written in the style of **BadInsert** will of course have many injection holes to try. An MVP colleague demoed such a program to me. He directed it at a couple of web sites, and it took the program maybe ten seconds to determine that a site was vulnerable to SQL injection.

It is very important to understand that the people who engage in this are not lonesome teenage boys who want to show off. No, the reason people spend their time on finding SQL injection holes spells m-o-n-e-y. If the database handles money by itself, the money can be made directly in the system. The money can also be made by stealing the information in the database and sell it. Or the intruder can use the SQL Server machine as an entrance to the rest of the network to infect all PCs in the organisation in preparation to attack a third party in a DDOS attack the intruder is paid to perform. Or the intruder may install ransomware and blackmail the site. Or...

Just to take one example from real life: Marriott announced in 2018 that almost 500 million customers' data had been leaked in a major data breach. And one of the means in the attack was, yes, SQL injection. You can search for *Marriott* and *sql injection* on your favourite search engine to find articles about the incident. Here is also a direct link to one of the articles I found.

While the biggest dangers are with web applications exposed on the Internet, this does not mean that you can ignore the risk for SQL injection in intranet applications or Windows clients. There may be malicious or disgruntled employees who want to steal data or just cause a mess in general. And there can be computers in the organisation that have been infected with malware that attempt to attack all machines they find.

So how do you protect yourself against SQL injection? In the xkcd strip, the mom talks about sanitising database inputs. What she presumably is alluding to is that the single quotes in the input string should be doubled. But that alone does not protect you against SQL injection; there are injection attacks that does employ the single quote. Therefore, it not uncommon to see suggestions that some characters should not be permitted input at all, but that is not a very fruitful way to go, since these characters may be part of perfectly legit input values. Just imagine that Brian O'Brien finds that he can't enter his name, but is told that it includes an illegal character!

No, there is a much better way to stop SQL injection and you have already seen it: parameterised statements. When input values are passed as parameters through **SqlParameterCollection** and ultimately to **sp_executesql**, SQL injection cannot happen, because there is no place to inject anything. That said, there is still one more precaution to take: the application should be running with a limited set of permissions. It is far too common for web applications to run with elevated permissions or even as **sa**. And that is very bad. After all, even if you have learnt by now how to avoid SQL injection, it only takes one junior programmer who has not and there is an SQL injection hole. Therefore, an application should never be granted more than membership in **db_datareader** and **db_datawriter**. That will limit the damage of an SQL injection hole. Even better, the application should only use stored procedures throughout. In that case, all you need to grant is EXECUTE permissions. (You can do this on schema or database level, no need to do it per procedure.)

Note: some readers may come with objections like "our application creates tables dynamically" etc so it must have elevated permissions. In that case, you should package these permissions inside stored procedures, and I describe this in detail my article *Packaging Permissions in Stored Procedures*. Applications should never run with elevated permissions, period!

3.2 SQL Injection in Dynamic SQL in Stored Procedures

When you work with dynamic SQL in a stored procedure, you should of course use **sp_executesql** with a parameterised statement to protect yourself against SQL injection in plain input values like search parameters etc. However, in the very most cases when you build an SQL string dynamically inside a stored procedure, it is because there is some input value that you cannot pass as a parameter, but which must be inserted into the SQL string, for instance a table name.

Here is an example. Charlie Brown is the DBA at a university where each student have their own little database where they can do whatever they like. Charlie wants to be helpful, so he schedules a job that loops through all tables in the student databases to defragment them. Here is the core of that script:

```
DECLARE @sql  varchar(MAX)

DECLARE table_cur CURSOR STATIC LOCAL FOR
   SELECT 'ALTER INDEX ALL ON [' + name + '] REBUILD ' FROM sys.tables

OPEN table_cur

WHILE 1 = 1
BEGIN
   FETCH table_cur INTO @sql
   IF @@fetch_status <> 0
      BREAK

   PRINT @sql
   EXEC(@sql)
END

DEALLOCATE table_cur
```

This is not a very good script. Charlie has added brackets around the table name, so the script will not fail if the student has created tables with spaces or other funny characters in the name. One flaw is that he has overlooked that tables do not have be in the **dbo** schema, but that is the small problem. One of the students at the university is Joe Cool. Here is a script to create his database:

```
CREATE LOGIN JoeCool WITH PASSWORD = 'CollegeKid'
CREATE DATABASE JoeCool
ALTER AUTHORIZATION ON DATABASE::JoeCool TO JoeCool
SELECT is_srvrolemember('sysadmin', 'JoeCool')
```

The final SELECT informs you that Joe is not a member of **sysadmin**. But that is exactly Joe's ambition, so in his database, he creates this:

```
USE JoeCool
go
CREATE TABLE tbl (a int NOT NULL)
CREATE TABLE [tbl]] REBUILD; ALTER SERVER ROLE sysadmin ADD MEMBER JoeCool --]
             (a int NOT NULL)
```

Yes, that last table has a funny name. Just as funny as Bobbie Tables in the xkcd strip above. At night, Charlie's Agent job runs, running with **sysadmin** permissions of course. Run the cursor batch above in the database **JoeCool**, and look at the statements (I've added a line break for legibility)

```
ALTER INDEX ALL ON [tbl] REBUILD
ALTER INDEX ALL ON
      [tbl] REBUILD; ALTER SERVER ROLE sysadmin ADD MEMBER JoeCool --] REBUILD
```

Now run this again:

```
SELECT is_srvrolemember('sysadmin', 'JoeCool')
```

Joe is now **sysadmin**.

Look at how Joe constructed his table name which I print out for clarity: **tbl] REBUILD; ALTER SERVER ROLE sysadmin ADD MEMBER JoeCool --**. (You can also run a query over **sys.tables** to see it.) He started with the name of another table in the database. He then added a right bracket, which he doubled in the CREATE TABLE script, since it was inside a bracketed name. The purpose of the bracket is to balance the left bracket in Charlie's script. Joe also added REBUILD to make the command syntactically complete. He then added the command he wanted to run, and at the end he added comment characters to kill the last part of Charlie's command.

Note: the command ALTER SERVER ROLE that Joe uses was introduced in SQL 2012. You can use **sp_addsrvrolemember** for the same trick on SQL 2008 and earlier.

There is a very simple change that Charlie can make to his script to stop Joe's attack. Before we look at it, let's first deprive Joe of his membership in **sysadmin**:

```
ALTER SERVER ROLE sysadmin DROP MEMBER JoeCool
```

Now change the SELECT statement in the cursor to read:

```
SELECT 'ALTER INDEX ALL ON ' + quotename(name) + ' REBUILD ' FROM sys.tables
```

Run the cursor again. This is the output (again with a line break to fit the page width):

```
ALTER INDEX ALL ON [tbl] REBUILD
ALTER INDEX ALL ON
      [tbl]] REBUILD; ALTER SERVER ROLE sysadmin ADD MEMBER JoeCool --] REBUILD
```

At first glance, it may seem identical to the above. However:

```
SELECT is_srvrolemember('sysadmin', 'JoeCool')
```

now returns 0. Joe was not promoted to **sysadmin**. If you look closer you can see why: the right bracket in Joe's table name is now doubled so the statement now achieves Charlie's goal: to rebuild the index on the table with the funny name. That is the purpose of the **quotename** function. Before we look closer at this important function, I like to make a comment on the example as such. Normally, you may think of a dynamic table name as something which comes in through a procedure, and which ultimately may come from the outside of the server. But as you saw this example, the attacker can also come from within the SQL Server instance.

3.3 Quotename and quotestring

Whenever you build dynamic SQL statements where a variable holds the name of something, you should use **quotename**. **quotename** returns its input surrounded by square brackets, and any right brackets are doubled as seen here:

```
SELECT quotename('a'), quotename('[a]')
```

The output is:

```
[a] [[a]]]
```

In this way, you protect yourself from SQL injection in metadata names. When you receive a table name as a parameter in a stored procedure, there are some more precautions that are more related to interface design than SQL injection as such, so we will look at this later in the style-guide chapter.

quotename accepts a second parameter that permits you to specify an alternative delimiter. Here are some examples to illustrate this:

```
SELECT quotename('Brian O''Brien', '''')      -- 'Brian O''Brien'
SELECT quotename('double (") quote', '"')     -- "double ("") quote"
SELECT quotename('(right) paren', '(')        -- ((right)) paren)
SELECT quotename('{brace} position', '}')     -- {{brace}} position}
SELECT quotename('right <angle>', '<')        -- <right <angle>>>
SELECT quotename('back`tick', '`')            -- `back``tick`
SELECT quotename('no $ money', '$')           -- NULL
```

In all cases, the closing delimiter is doubled. As you can see, for the paired delimiters, you can pass either the left or the right delimiter, but it is always the closing delimiter that is doubled. As the last example suggests, you cannot pass any character as the delimiter, but all the supported ones are shown above. Of these, the most important is the first one, which you need to use if you want to incorporate a string value in your statement. As an example, here is a batch that creates a database with the name set dynamically:

```
DECLARE @dbname   sysname = 'My New Database',
        @datadir  nvarchar(128),
        @logdir   nvarchar(128),
        @sql      nvarchar(MAX)
SET @datadir = convert(nvarchar(128), serverproperty('InstanceDefaultDataPath'))
SET @logdir = convert(nvarchar(128), serverproperty('InstanceDefaultLogPath'))

SELECT @sql = 'CREATE DATABASE ' + quotename(@dbname, '"') + '
    ON (NAME     = ' + quotename(@dbname, '''') + ',
        FILENAME = ' + quotename(@datadir + @dbname + '.mdf', '''') + ')
    LOG ON (NAME     = ' + quotename(@dbname + '_log', '''') + ',
            FILENAME = ' + quotename(@logdir + @dbname + '.ldf', '''') + ')'
PRINT @sql
EXEC(@sql)
```

This was the statement generated on my machine:

```
CREATE DATABASE "My New Database"
    ON (NAME     = 'My New Database',
        FILENAME = 'S:\MSSQL\SQLXI\My New Database.mdf')
    LOG ON (NAME     = 'My New Database_log',
            FILENAME = 'S:\MSSQL\SQLXI\My New Database.ldf')
```

Note: this example does not run on SQL 2008 and earlier. These two parameters to **serverproperty** were introduced in SQL 2012.

Observe that what I pass to **quotename** is the full path to the two database files; that is, string concatenation occurs *inside* **quotename**. For the sake of the example, I used double quotes to delimit the database name, rather than the more common square brackets. (The latter is Microsoft-specific, whereas the double quote is the ANSI standard). As for the other delimiters you can use with **quotename**, the situations where you get to use them are few and far between.

quotename is a very convenient function, but there is a trap that you need to beware of. As the name suggests, **quotename** is intended to be used with object names. Its input is limited to 128 characters (which is the maximum length of an identifier in SQL Server), as seen in this example:

```
DECLARE @x nvarchar(255) = replicate('''', 128)
SELECT quotename(@x, '''')
SET @x = replicate('''', 129)
SELECT quotename(@x, '''')
```

The first SELECT returns a long list of single quotes (more precisely 258 of them!), but the second returns NULL. Thus, as long as you only use **quotename** to delimit identifiers in brackets, you are safe. But when you use **quotename** to delimit string values in single quotes, you need to consider that the value may exceed 128 characters in length. The CREATE DATABASE example above is certainly vulnerable to this risk. The return type for the two **serverproperty** parameters is **nvarchar(128)** and this is also the definition of **sysname**. Thus, the full file name can be up to 260 characters long.

If you want to eliminate this risk, you can use the user-defined functions **quotestring** and **quotestring_n**. Both accept **nvarchar(MAX)** as input and return the same data type. They wrap their input in single quotes, and double any single quotes within the strings. **quotestring_n** also tacks on an N to make the returned string an **nvarchar** literal. With **quotestring**, you need to cater for this yourself. They are not a built-in, but listed below:

```
CREATE FUNCTION quotestring(@str nvarchar(MAX)) RETURNS nvarchar(MAX) AS
BEGIN
   DECLARE @ret nvarchar(MAX),
           @sq  nchar(1) = ''''
   SELECT @ret = replace(@str, @sq, @sq + @sq)
   RETURN(@sq + @ret + @sq)
END
go
CREATE FUNCTION quotestring_n(@str nvarchar(MAX)) RETURNS nvarchar(MAX) AS
BEGIN
   DECLARE @ret nvarchar(MAX),
           @sq  nchar(1) = ''''
   SELECT @ret = replace(@str, @sq, @sq + @sq)
   RETURN('N' + @sq + @ret + @sq)
END
```

We will see examples of using these functions throughout in this atricle.

You may ask: when should I use **quotename** (or **quotestring**)? *Every time* you insert an object name or a string value from a variable into your SQL string, without any exception. Take the database example above: the two paths are not likely attack vectors, since you need to be **sysadmin** or a Windows administrator to change them. But the database name may come from an untrusted source (like our college kid Joe Cool) and contain something malicious. And even if all this is hypothetic to you, there is still argument of general robustness: if any of the names would happen to include a single quote, why should the command fail with a syntax error?

3.4 The Importance of nvarchar

This is an SQL injection trap that I was unaware of myself, until I by chance was looking at [Remus Rusanu's blog] for something else as I was working on this article.

There is one more flaw in Charlie Brown's script above, and which Joe Cool exploits by creating this table:

```
CREATE TABLE [tbl〛 REBUILD; ALTER SERVER ROLE sysadmin ADD MEMBER JoeCool --]
             (a int NOT NULL)
```

That funny character after the last character in **tbl** is the Unicode character U+301B, Right White Square Bracket. Night comes, and Charlie's enhanced script using **quotename** runs and this is printed (with line-breaks added):

```
ALTER INDEX ALL ON [tbl] REBUILD
ALTER INDEX ALL ON
      [tbl]] REBUILD; ALTER SERVER ROLE sysadmin ADD MEMBER JoeCool --] REBUILD
ALTER INDEX ALL ON
      [tbl] REBUILD; ALTER SERVER ROLE sysadmin ADD MEMBER JoeCool --] REBUILD
```

Note that the funny double-bracket character is now a regular right bracket. If you run

```
SELECT is_srvrolemember('sysadmin', 'JoeCool')
```

you see that Joe Cool is again **sysadmin**.

The reason this happened is that Chaiile was sloppy in his script when he declared the **@sql** variable:

```
DECLARE @sql  varchar(MAX)
```

It's **varchar**, not **nvarchar**. When data is converted from **nvarchar** to **varchar**, any character in the Unicode string that is not available in the smaller character set used for **varchar** is replaced with a fallback character. And in this case, the fallback character is a plain right bracket. **quotename** did not help here, because **quotename** worked on the input from **sys.tables** which is **nvarchar** and the conversion to **varchar** happened after **quotename** had done its job.

Fix the declaration of **@sql** and take Joe out of **sysadmin**:

```
ALTER SERVER ROLE sysadmin DROP MEMBER JoeCool
```

Change the variable declaration and run Charlie's script again. This time the output is:

```
ALTER INDEX ALL ON [tbl] REBUILD
ALTER INDEX ALL ON
      [tbl]] REBUILD; ALTER SERVER ROLE sysadmin ADD MEMBER JoeCool --] REBUILD
ALTER INDEX ALL ON
      [tbl〛 REBUILD; ALTER SERVER ROLE sysadmin ADD MEMBER JoeCool --] REBUILD
```

The right white square bracket is still there and **is_srvrolemember** informs us that Joe is not **sysadmin**.

This character is not the only one, but there are more Unicode characters that can be used for such exploits. For instance, in his blog Remus mentions Modifier Letter Apostrophe, U+02BC, which gets replaced by a single quote when the string is cast to **varchar**.

We can also use this example to chalk one up for **sp_executesql**. Since **sp_executesql** requires you to use **nvarchar** for your **@sql** variable, you cannot open up this type of hole; it can only happen to you if you use EXEC().

Note: While the main purpose of these examples is to demonstrate that SQL injection can sneak in where you may not expect it, the major mistake of Charlie is another one: he is running with elevated permissions when he shouldn't. Whenever you as a server-level DBA run code in a user database you cannot trust, you should bracket your commands in EXECUTE AS USER = '**dbo**' and REVERT. This sandboxes you into the current database, and you no longer have permissions on server level, and a user like Joe cannot exploit them to become **sysadmin**, or anything else he cannot do himself, assuming that he has **db_owner** rights in the database. The SQL injection attacks we have seen is not the only type of attack that Joe can perform. For instance, he could add a DDL trigger that is fired on ALTER INDEX and which adds him to **sysadmin** when Charlie's script runs.

3.5 Other Injection Holes

In all examples so far, the attack vector has been where a variable which is supposed to hold a single token has been concatenated to the SQL string. Since it is a single token, protection with parameterisation and **quotename** is simple.

If you have variables that are intended to hold multiple tokens, it can be very difficult to protect yourself against SQL injection. Here is one example, which I still see far too often:

```
SELECT @list = '1,2,3,4'
SELECT @sql = 'SELECT * FROM tbl WHERE id IN (' + @list + ')'
```

How do you ensure that **@list** really is a comma-separated list of values and not something like `'0) DROP DATABASE abc --'`? The answer is this particular case is that you don't use dynamic SQL at all, but you should use a list-to-table function. See my short article *Arrays and List in SQL Server* for alternatives.

Another example is this: say that you have a stored procedure that returns a result set and you are already using dynamic SQL to produce that result set. You decide to beef it up by adding a parameter to specify the sort order. It may seem as simple as:

```
CREATE OR ALTER PROCEDURE my_dynamic_sp .....
                          @sortcols nvarchar(200) = NULL AS
   ...
   IF @sortcols IS NOT NULL
      SELECT @sql +=  ' ORDER BY ' + @sortcols
   ...
```

But as you understand by now, this procedure is wide open to SQL injection. In this case, you may argue that you have it all under control: you are showing the column names to the user in a dropdown or some other UI device, and there is no place where the user can type anything. That may be true today, but it may change two years from now, when it is decided to add a web interface, and all of a sudden the sort columns come as parameters in a URL (which the user of course has all the powers to manipulate). When you are writing a stored procedure for application use, you should never make any assumptions about the client, even if you write the client code as well. Least of all when it comes to serious security risks as SQL injection.

There are several ways to skin this cat. You could parse **@sortcols** and then apply **quotename** on the individual columns. Personally, I think I would prefer to use a three-column table-valued parameter (one column for the position, one for the column name and a bit column for ASC/DESC). Or I might prefer to entirely decouple the client from the column names and do something like this:

```
SELECT @sql += CASE @sortcol1
                    WHEN 'col1' THEN 'col1'
                    WHEN 'col2' THEN 'col2'
                    ...
                    ELSE             'col1'   -- Need to have a default.
                END + ' ' + CASE @isdesc1 WHEN 0 THEN 'ASC' ELSE 'DESC' END
```

If **col2** would to be renamed to something else, the client would not be affected.

And to generalise this: whenever you think of passing variables with multi-token SQL text to your stored procedure in an application, you need to work out a solution for your stored procedure so that is not open to SQL injection. This is by no means simple, since you have to parse fragments of SQL code, something which T-SQL is not well-equipped for. There can be all reason to reconsider your approach entirely. For instance, if you have the idea of passing a full-blown WHERE clause, you should probably not use a stored procedure at all, but construct all SQL in the client. (Passing a WHERE clause from client to server is not only bad because of the risk for SQL injection, but also because it introduces a tight coupling between client and server that will be a maintenance headache in the long run.) There is no reason to use stored procedures for the sake of it. There are scenarios which are very dynamic where you want to give users the ability to retrieve data with a high degree of flexibility. Such scenarios are best solved client-side, and in such solutions, the SQL code is likely to be chopped up in small fragments that are pieced together. Obviously, you should still protect yourself against SQL injection, and you should still use parameters for values in WHERE clauses. Names of columns and tables should never come from text fields or URLs, but from sources you have full control over.

So far application code. For a pure DBA utility you can be permit yourself to be a little more liberal. To take one example, in my article *Packaging Permissions in Stored Procedures*, I present a stored procedure **GrantPermsToSP** of which one parameter is a TVP with permissions to be inserted into a GRANT statement. I make no validation of this parameter, and the procedure is indeed open for SQL injection. However, as it is intended to be called from SSMS by persons who are in the **db_owner** role, there is nothing they can gain by exploiting the injection hole – they cannot run more powerful commands through the procedure than they can run directly themselves. And that is the quintessence: SQL injection is a threat, when it permits users to elevate their permissions beyond from what they can do on their own from SSMS or similar. (Keep in mind that for web users who have no direct access to SQL Server at all, this means that any command they can inject is an elevation.) From this follows that if you sign your utility procedure with a certificate to package permissions with the procedure to permit low-privileged users to run some privileged command in a controlled way (and that is exactly what you use **GrantPermsToSP** for), you must absolutely make sure that you don't have any SQL injection hole.

---

4. Performance

You may ask if there are any performance differences between using dynamic SQL and static SQL in stored procedures. The answer is that as long as you observe a few guidelines when you compose your dynamic SQL, there is hardly any difference at all. (Well, if you produce a 2000-line beast in your client and pass it to SQL Server, you should probably consider moving it to a stored procedure, as there is a cost for passing all that code across the wire.)

When you construct your dynamic SQL there are two things you need to observe to avoid wreaking havoc with performance. One of them you have already learnt. That is, you should always parameterise your statements and not inline simple parameter values for several reasons:

- It makes the code easier to read.
- It makes it easier to deal with dates.
- It makes it easier to deal with input like Brian O'Brien.
- You protect yourself against SQL injection.

Now you will learn one more reason: it is better for performance. Not so much for your own individual query, but for the server as a whole. This is related to how SQL Server caches query plans. For a stored procedure, SQL Server looks up the query plan in the cache by the name of the stored procedure. A batch of dynamic SQL does not have a name. Instead, SQL Server computes a hash value from the query text and makes a cache lookup on that hash. The hash is computed on the query text as-is without any normalisation of spaces, upper/lower, comments etc. And most importantly, not of parameter values inlined into the string. Say that you have something like this:

```
cmd.CommandText = @"SELECT ...
                   FROM   ...
                   JOIN   ...
                   WHERE col1 = " + IntValue.Tostring();
```

Say now that this command text is invoked for **IntValue** = 7, 89 and 2454. This results in three different cache entries. Now imagine that the query has several search parameters and there is a multitude of users passing different values. There will be a lot of cache entries, and this is not good for SQL Server. The more entries there are, the bigger the risk for hash collisions. (That is, two different query texts have the same hash value, whereupon SQL Server needs to compare the query texts directly.) On top of that, the cache starts to consume a lot of memory that could be better used elsewhere. Eventually, this can lead to plans to be kicked out of the cache, so that reoccurring queries needs to be recompiled. And last but not least: all this compilation will take a big toll on the CPU.

I like to make you aware of that when SQL Server computes the hash value, this includes the parameter list. Thus, these two calls will result in two cache entries:

```
EXEC sp_executesql N'SELECT COUNT(*) FROM Customers WHERE City = @city',
                   N'@city nvarchar(6)', N'London'
EXEC sp_executesql N'SELECT COUNT(*) FROM Customers WHERE City = @city',
                   N'@city nvarchar(7)', N'Abidjan'
```

Note the difference in length of the declaration of the **@city** parameter.

And this is what you get if you in .NET define the parameter in this way:

```
cmd.Parameters.Add("@strval", SqlDbType.NVarChar).Value = city;
```

When you leave out the length, .NET will set the length from the actual value of **city**. So this is why you should always set the length explicitly for string and binary parameters. And stay away from **AddWithValue**.

Note: Being an honest person, I need to confess that I have lied a bit here. For very simple queries, SQL Server will always employ auto-parameterisation, so that constants are replaced by parameter placeholders. The INSERT statement in the **clientcode** program is an example of such a query. There is also a database setting, forced parameterisation, under which SQL Server replaces about any constant in a query with a parameter. This setting is indeed intended to be a cover-up for poorly written applications that inline parameter values into SQL strings. But you should never make any assumption of that auto-parameterisation will save the show, but you should always use parameterised statements.

In this note I should also point out that there is a configuration option **optimize for ad hoc workloads**. When this is set, the plan for an unparameterised query string is not cached the first time it appears, but only a shell query is stored in the cache. The plan is cached only if the exact same query string appears a second time. This reduces the memory footprint, but the hash collisions will still be there. It's generally considered best practice to have this option turned on, but with a well-written application, it should not really matter.

Note that this applies to values that comes as input from the user or some other source. A query might have a condition like

```
AND customers.isactive = 1
```

That, the query is hard-wired to return only active customers. In this case, there is absolutely no reason to make the 1 a parameter.

There is one more measure you need to take to reduce cache-littering: always schema-qualify your tables, even if all your tables are in the **dbo** schema. That is, you should write:

```
cmd.CommandText = @"SELECT ...
                    FROM    dbo.sometable
                    JOIN    dbo.someothertable ON ...
                    WHERE   col1 = @id"
```

The reason for this is as follows: Assume two users Jack and Jill. Jack has the default schema **Jack** and Jill has the default schema **Jill**. If you leave out **dbo** and say only `FROM sometable`, SQL Server needs to consider that at any point in time there may appear a table named **Jack.sometable** or **Jill.sometable**, in which case Jack's and Jill's queries should go against these respective tables. For this reason, Jack and Jill cannot share the same cache entry, but there has to be one cache entry each for them. If there are many users, this can lead to many cache entries for the same query.

Observer that this applies to everything you put in a query that belongs to a schema: views, table-valued functions etc. All of this should be referred to by two-part notation. If you miss a single one, the cache entry cannot be shared by users with different default schemas.

Note: When you create a user with CREATE USER, the default schema is set to **dbo**, so there is a fair chance that all users have the same default schema, and thus can share cached query plans even if you leave out the schema. But there is no reason to assume that this is always true. There are older commands to create users, and they will also create a schema for the user and make that the default schema.

In this section I have only used examples in client code, but everything I have discussed applies to dynamic SQL created in a stored procedure as well.

- -

5. Working with Dynamic SQL – a Style Guide

You have learnt the basic commands for dynamic SQL, and you have been warned about SQL injection. Now is the time to learn more about dynamic SQL from a practical standpoint. You may think that writing dynamic

SQL is a simple task: build strings, how difficult can it be? From experience, I can tell you that it is more difficult to get it right that it may seem initially.

Writing good dynamic SQL is not only about skill. No, maybe the most important virtue is *discipline*. If you are consistent and careful when you write your dynamic SQL, you will have fewer bugs and your code will be easier to read and maintain. On the other hand, poorly written code that generates dynamic SQL tends to be very difficult, not to say impossible, to read and understand. The generated code can easily get buried in the T-SQL code that generates it with all its single quotes and plusses. Generally, the fewer of these you have, the more readable your code will be, and the less is the risk for errors. You have already learnt the most important device: use parameterised statements. But since you cannot parameterise everything, that is not enough. In this chapter I will give some tips on how to improve your dynamic SQL from different angles: how to make it more readable, easier to troubleshoot, more robust and more secure.

5.1 Start Out Static

Before you start writing dynamic SQL, you need to have a good understanding of what the query that you are aiming to generate is going to look like. The best way is to start with writing a fully static query, where you temporarily hardcode the parts you want to be dynamic. Once you have the static query working, you can use that a starting point to write the code that builds the dynamic query. If you try to solve all at once, you may find yourself in a situation where you don't know if it is the query logic as such that is wrong, or whether you have built the query in the wrong way.

5.2 The Necessity of Debug Prints

If you are new to dynamic SQL the first thing you need to learn is to use debug prints. All stored procedures that deals with dynamic SQL, should include a final parameter **@debug**:

```
@debug bit = 0 AS
```

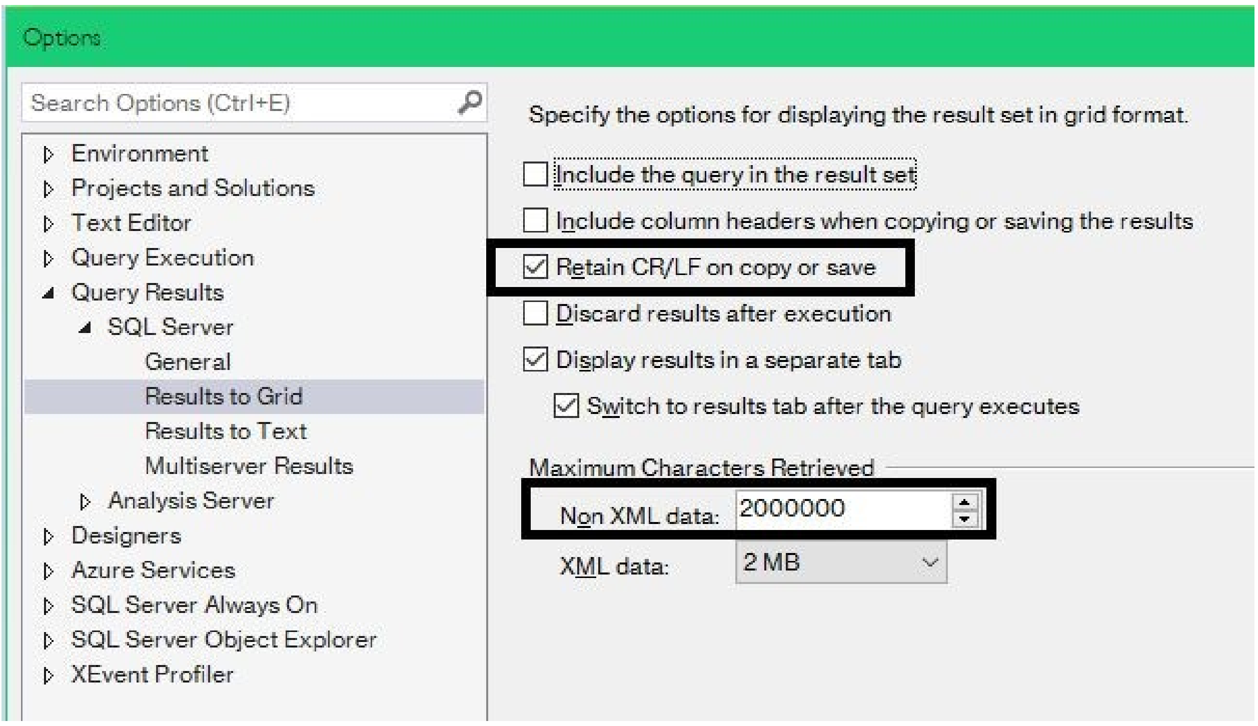
And in everywhere in the code where you are about to run some dynamic SQL, you should have:

```
IF @debug = 1 PRINT @sql
EXEC sp_executesql @sql, @params, @par1, ...
```

This is absolutely essential, because every once in a while your SQL string will not do what you intended to, and it may even fail to even compile. If you get an error message, and you don't see what the string looks like, you are in a completely dark room. On the other hand, if you see the SQL code, the error may be immediately apparent.

If your SQL string is very long, the statement may come out as a truncated. The PRINT statement prints at most 8000 bytes, so with **nvarchar(MAX)** for your **@sql** variable, you will only see up to 4000 characters.

One way to avoid this is to use SELECT instead of PRINT. That gives you the string in a grid cell in SSMS, and when you copy it to a text window, you may find that it is all one line. Thankfully, modern versions of SSMS has a setting to control this. In SSMS, go to Tools->Options and check the setting *Retain CR/LF on copy or save* highlighted as in this picture:



Specify the options for displaying the result set in grid format.

☐ Include the query in the result set
☐ Include column headers when copying or saving the results
☑ Retain CR/LF on copy or save
☐ Discard results after execution
☑ Display results in a separate tab
☑ Switch to results tab after the query executes

Maximum Characters Retrieved
Non XML data: 2000000
XML data: 2 MB

You can also see that I have increased the maximum for *Non XML Data* to two million (the default is 65536) to avoid truncation with very long strings. Observe if you change these settings, none of them affect currently open query windows, but only windows you open after the change.

Note: To see the max above 65536, you need to have SSMS 18.2 or later. The checkbox for retaining CR/LF was introduced in SSMS 16 and it is unchecked by default. If you don't see this setting, you have an older version of SSMS. In SSMS 2012 and 2014, CR/LF are always retained, and in SSMS 2005 and 2008, they are always replaced by a single space. In any case, since SSMS is a free download, there is little reason to use an old version.

While this works, it is not without issues:

- If you frequently copy data to Excel, you may prefer to keep the *Retain* setting unchecked. (Because if there are line breaks in the data, the mapping to cells in Excel will be messed up.)
- There are situations where extra result sets from your stored procedure could mess things up, for instance if you use INSERT-EXEC, so you would prefer to get the code printed to the *Messages* tab in SSMS.
- If your procedure produces several batches of dynamic SQL, it may be more practical to have them all collected in the *Messages* tab, rather than having to copy multiple result sets from the query grid.

I got a suggestion from Jonathan Van Houtte that addresses the first point (but the the other two) which makes use of the XML capabilities in SQL Server and SSMS:

```
SELECT @sql AS [processing-instruction(x)] FOR XML PATH('')
```

In the grid cell, you get a clickable XML string and when you click on it, it opens in a new window where you see the SQL text with the CR/LF retained. The code is wrapped in a `<?x` tag, but apart from that, there is no mutilation due to the XML. Only if the SQL string would happen to include the sequence `?>`, you could get problems.

My co-worker Jan Swedin came up with the ultimate solution to this problem: a stored procedure that breaks up the text in chunks of 8000 bytes, taking care that each chunk ends with a line break, and then he runs a PRINT on every chunk. Here is an adapted version of his procedure:

```
CREATE OR ALTER PROCEDURE SystemPrint @str nvarchar(max) AS
SET XACT_ABORT, NOCOUNT ON;
DECLARE @noofchunks int,
        @chunkno    int = 0,
        @pos        int = 0, -- substring start posotion
        @len        int;     -- substring length

SET @str = replace(@str, nchar(13), '');
SET @noofchunks = ceiling(len(@str) / 4000.0);

WHILE @chunkno < @noofchunks
BEGIN
   SET @len = 4000 - charindex(nchar(10) COLLATE Latin1_General_BIN2,
                               reverse(substring(@str, @pos, 4000)));
   PRINT substring(@str, @pos, @len);
   SET @chunkno += 1;
   SET @pos += @len + 1; -- accumulation + LF
END
```

Note: there is a known issue with this procedure. If the length of SQL string is close to a full chunk, for instance 7998 characters, the last few characters in the string may not print. It is left as an exercise to the reader to understand how this can happen and fix the issue.

No matter which of these methods you go for, always have a debug print of your dynamic SQL. And, yes, I mean ALWAYS!

5.3 Error Messages and Line Numbers

When you work with dynamic SQL, you will ever so often encounter error messages. Sometimes the error is related to the code that generates the dynamic SQL, sometimes it comes from the dynamic SQL itself. In either case, it can be a compilation error or a run-time error. It goes without saying that you need to understand the origin of the error, so that you start looking at the right piece of code. Unfortunately, this is something which is more complicated than it has to be, due to a feature in recent versions of SSMS.

For this section we will work with the task of creating an SQL login, a database user for that login and add the user to the role **PlainUsers**. Input for the operation is the login name and the password. We will first look at the situation where the dynamic SQL is generated inside a stored procedure. Copy this code, which has a number of errors in it, into an empty query window.

```
USE tempdb
go
IF NOT EXISTS (SELECT * FROM sys.database_principals WHERE name = 'PlainUsers')
   CREATE ROLE PlainUsers
go
CREATE OR ALTER PROCEDURE add_new_user @name     sysname,
                                       @password nvarchar(50),
                                       @debug    bit = 0 AS
   DECLARE @sql nvarchar(MAX)
   SELECT @sql = 'CREATE LOGIN ' + quotename(@name) + '
                  WITH PASSWORD = ' + quotename(@password, '''') + ' MUST_CHANGE,
                       CHECK_EXPIRATION
              CREATE USER ' + quotename(@name) + '
              ALTER ROLE PlainUsers ADD MEMBER ' + quotename(@name)
   IF @debug = 1
      PRINT @sql
   EXEC sp_exectesql @sql
go
EXEC add_new_user 'nisse', 'TotaL1y#and0m', 1,
```

Note: If you are running these examples on SQL 2008 or earlier, you may observe that the code uses the syntax ALTER ROLE ADD MEMBER which was added in SQL 2012. However, since we are exploring error messages, this should not be of much practical importance to you.

The first error message we encounter when we try this piece of code has no relation to dynamic SQL as such, but serves to highlight a behaviour in modern versions of SSMS that is important to understand for this section. This is the output when you run the script in SSMS 2014 or later.

```
The module 'add_new_user' depends on the missing object 'sp_exectesql'. The module will still
be created; however, it cannot run successfully until the object exists.
Msg 102, Level 15, State 1, Line 19
Incorrect syntax near ','.
```

If you double-click the error message, you will find that you end up on the line for the EXEC statement after the procedure, and indeed there is a trailing comma that should not be there. And if you look in the status bar, you will find that it says *Ln 19*, which agrees with the error message. However, this is not the error number that SQL Server reports. Keep in mind that SSMS sends one batch, as defined by the `go` separator, at a time to SQL Server, which has no knowledge about your query window. The call to **add_new_user** is a single-row batch, so the actual line number reported by SQL Server is Line 1. This is also the line number you would see if you were to run this in SSMS 2012 or earlier. You can convince yourself of this fact by saving the script to a file and run it through SQLCMD, which make no adjustments. Why this important to understand will prevail in just a few moments.

Remove the offending comma (or replace it with a semicolon) and try again. This time, the output is:

```
The module 'add_new_user' depends on the missing object 'sp_exectesql'. The module will still
be created; however, it cannot run successfully until the object exists.
CREATE LOGIN [nisse]
                    WITH PASSWORD = 'TotaL1y#and0m' MUST_CHANGE,
                         CHECK_EXPIRATION
                CREATE USER [nisse]
                ALTER ROLE PlainUsers ADD MEMBER [nisse]
Msg 2812, Level 16, State 62, Procedure add_new_user, Line 12 [Batch Start Line 18]
Could not find stored procedure 'sp_exectesql'.
```

(The part *Batch Start Line 18*, appears only in SSMS 16 and later. If you are using SSMS 2014 and earlier, you will not see that part.) Observe here that the error message includes the procedure name **add_new_user**. This is very important. This tells us that the error occurred during compilation or execution of the stored procedure. Because the name is present, the error cannot come from the dynamic SQL. (Recall that the dynamic SQL is a nameless stored procedure of its own.) SSMS sees the procedure name, and realises that there is little point in modifying the line number from SQL Server, and the line number shown here is really line 12 in the stored procedure, not line 12 in the query window. (In this particular example, the procedure happens to be present in the query window, but many times you will call stored procedures not visible in your current window.)

Fix the typo and re-run the procedure. Now we get this output in SSMS 16 and later.

```
CREATE LOGIN [nisse]
                    WITH PASSWORD = 'TotaL1y#and0m' MUST_CHANGE,
                         CHECK_EXPIRATION
                CREATE USER [nisse]
                ALTER ROLE PlainUsers ADD MEMBER [nisse]]
Msg 156, Level 15, State 1, Line 22
Incorrect syntax near the keyword 'CREATE'.
```

There is no procedure name in the error message, so the error cannot come directly from within **add_new_user**. Could it come from the dynamic SQL? Or could it come from something directly in the query window? We could try to double-click the error message and see where we land. If you try this, you may not be able to make much sense of it, but if you try it multiple times with the cursor placed on different lines, you will eventually realise that the cursor stays where you were. You may also get a message box from SSMS about something being out of bounds. Whatever, if you check more closely, you realise that the error message says line 22, but there are only 19 lines in the script. And the dynamic SQL is only five lines of code, so where did that 22 come from?

Above, SSMS was helpful and modified the line number from SQL Server to a line number in the query window, and this is what is happening here as well. But what's the point with modifying the line number from the dynamic SQL? Obviously, we want to see the actual line number as reported by SQL Server. Trying to map that to a line number in the query window seems just silly. The answer is that there is no way for SSMS to discern whether the error message comes from the SQL code directly present in the query window, or some batch of dynamic SQL executed from that window. The error information provided by SQL Server has no such details, so SSMS just assumes that the error comes from something directly present in the window.

This leads to a very awkward situation. In this example, it is maybe not that difficult, since the dynamic SQL is only five lines, but say that you generate a larger batch of dynamic SQL consisting of 50 to 100 lines of code. In this case, you really want to know the line number within the batch, so that you can find where error really is.

The process is as follows. You first find the start of the batch you actually executed, which in this case is this line:

```
EXEC add_new_user 'nisse', 'TotaL1y#and0m', 1
```

The status bar informs us that this is line 19. Had the error been on the first line of dynamic SQL, SQL Server would have reported Line 1 and SSMS would have said 19. Now it was 22, and 22-19+1 = 4. Thus, according to SQL Server, the error is on line 4. The actual error is that the correct syntax for the CHECK_EXPIRATION clause is CHECK_EXPIRATION = ON, but SQL Server does not detect the error until it sees the CREATE on the next line. If you add the missing syntax and try again, you will find that the batch executes successfully. (If you have SQL 2012 or later, as noted above.)

Here I had a stored procedure, and therefore it was simple to tell whether the error was from the code that generated the dynamic SQL, or the dynamic SQL itself. But say that we want to make this a plain script, because we want to use it on different servers or databases on different occasions. Paste the script below into to the same query window as the procedure above, adding a `go` separator after call to **add_new_user**, and then run what you pasted:

```
DECLARE @name     sysname      = 'linda',
        @password nvarchar(50) = 'UlTr@-TOp_$ecre1',
        @debug    bit = 1

DECLARE @sql nvarchar(MAX)
SELECT @sql = 'CREATE LOGIN ' + quotename(@name) + '
                  WITH PASSWORD = '''  quotename(@password, '''') + ' MUST_CHANGE,
                       CHECK_EXPIRATION = ON
               CREATE USER ' + quotename(@name) + '
               ALTER ROLE PlainUsers ADD MEMBER ' + quotename(@name)
IF @debug = 1
   PRINT @sql
EXEC sp_executesql @sql
```

This is the output:

```
Msg 102, Level 15, State 1, Line 27
Incorrect syntax near 'quotename'.
```

Is this message coming from the dynamic SQL or from the script itself? In this example there is no procedure name to guide us. But there are some clues. One clue is the appearence of **quotename** in the error message; this suggests that it's an error in the script itself, as **quotename** does not seem to be used within the dynamic SQL. Another clue is that the contents of **@sql** is not printed, which it would be if execution had reached the line that invokes the dynamic SQL. We can get yet one more clue by double-clicking the error message to see if we hit something matches the error message. This time, we end up on this line:

```
                  WITH PASSWORD = '''  quotename(@password, '''') + ' MUST_CHANGE
```

If we look closer, we may note that there should be a + between the sequence of single quotes and **quotename**. Insert the plus character and run again:

```
CREATE LOGIN [linda]
                 WITH PASSWORD = ''UlTr@-TOp_$ecre1' MUST_CHANGE,
                      CHECK_EXPIRATION = ON
               CREATE USER [linda]
               ALTER ROLE PlainUsers ADD MEMBER [linda]
Msg 102, Level 15, State 1, Line 22
Incorrect syntax near 'UlTr@'.
Msg 105, Level 15, State 1, Line 22
Unclosed quotation mark after the character string ' MUST_CHANGE,
               CREATE USER [linda]
                      CHECK_EXPIRATION = ON
                 ALTER ROLE PlainUsers ADD MEMBER [linda]'.
```

Since we see the SQL printed, that is a strong indication that the error is in the dynamic SQL. Since the text *UlTr@* appears in the message text, the text alone helps us to locate where the error is. But say that the situation is not that fortunate, but we need to translate the error number from SSMS to a line number in the dynamic SQL. Place the cursor on the first line in the batch, and in this example this is the line with the first DECLARE. This is line 21, and thus the line number that SQL Server reports is 22-21+1 = 2. If your batch has one or more blank lines before the DECLARE, then you should put the cursor on the first of these blank lines – ***if*** you included these blank lines when you selected the query text to execute. That is, the base number is the first line of what you selected in the query window.

Note that this highlights the importance of having those debug PRINTs. If you don't see the debug output, you can suspect that the error message is in the script itself, and conversely, if you see the debug output, the error is likely to be in the dynamic SQL. (Although we saw an exception from that rule with the misspelling of **sp_executesql** above.)

If you have a script that produces multiple batches of dynamic SQL, things can get really complicated. If you get an error message after the printout of a batch of dynamic SQL, the error may be in that batch – or it may come from the code that generates the next batch. You will have to use your common sense to find out what is going on. If you double-click the error message and the line you end up has no relation to the error message that is an indication of that the error is in dynamic SQL.

5.4 Spacing and Formatting

As I said earlier, an important virtue when you work with dynamic SQL is discipline, and this really comes into the picture when it comes to spacing. You need to take care so that different parts of your dynamic SQL don't run into each other. Here is an example where sloppiness is rewarded with an error message:

```
DECLARE @sql nvarchar(MAX)
DECLARE @tblname sysname = 'dbo.Customers'
SELECT @sql = 'SELECT CompanyName, ContactName, Address
               FROM' + @tblname +
              'WHERE City = @city'
PRINT @sql
EXEC sp_executesql @sql, N'@city nvarchar(15)', 'London'
```

The output is:

```
SELECT CompanyName, ContactName, Address
              FROMdbo.CustomersWHERE City = @city
Msg 102, Level 15, State 1, Line 36
Incorrect syntax near '.'.
```

The FROM and the WHERE have been glued with the table name because of the missing spaces in the SQL string. (As noted above, the line number depends on exactly where in the window you pasted the command. I ran the above in a window I already had some code in, and SSMS reported the error to be on line 36.)

You should make it a habit to make sure that there is always white space before and after keywords when you split up the string to bring in the contents of a variable or whatever. White space includes newline, and since T-SQL permits string literals to run over line boundaries, you should make use of this. This is a better example of the above:

```
DECLARE @sql nvarchar(MAX)
DECLARE @tblname sysname = 'dbo.Customers'
SELECT @sql = 'SELECT CompanyName, ContactName, Address
               FROM ' + @tblname + '
               WHERE City = @city'
PRINT @sql
EXEC sp_executesql @sql, N'@city nvarchar(15)', 'London'
```

The SQL batch now runs successfully. This is the SQL string that is printed:

```
SELECT CompanyName, ContactName, Address
```

```
                 FROM dbo.Customers
                 WHERE City = @city
```

The indentation is a little funny, but the code is perfectly readable.

If you are coming from a language like Visual Basic which does not permit string literals to run over line breaks, you may prefer something like this:

```
DECLARE @sql nvarchar(MAX)
DECLARE @tblname sysname = 'dbo.Customers'
SELECT @sql = 'SELECT CompanyName, ContactName, Address ' +
              ' FROM ' + @tblname +
              ' WHERE City = @city '
PRINT @sql
EXEC sp_executesql @sql, N'@city nvarchar(15)', 'London'
```

While this code as such certainly is fairly easy to read, I still strongly recommend against this for more than one reason. One is that the dynamic SQL will be one single long line, so if there is an error message, it will always be reported to be on line 1 (after you have applied the transformations in the previous section). If the error message is cryptic and the SQL text is long, you can be quite in the dark about where the error really is. A more general reason is that in my experience, the boundaries between all these plusses and quotes is a danger area where errors easily creep in because we accidently leave out one or the other. So the fewer we have of them, the better.

As I said in the beginning of this chapter: when you are about to build something with dynamic SQL, first write a static query where you have sample values for whatever you need to have dynamic at run-time. This serves two purposes:

- You make sure that you have the syntax of the query as such correct.
- You get the formatting you like of the query.

I like to stress that the formatting of the dynamic SQL is important for two reasons:

1. If there is a compilation error in the code, the line number helps you to find where the error is. (Never mind that you have to go through the hoops I discussed in the previous section.)
2. The code executes, but does not produce the expected results. It certainly helps if the query is formatted in a way you are used to when you are trying to understand what is wrong with it.

Sometimes you use several statements to build your dynamic SQL, because you are picking up different pieces from different places. A prime example is when you write something to support dynamic search conditions. A typical passage in such a procedure could be:

```
IF @orderid IS NOT NULL
   SET @sql += ' AND O.OrderID = @orderid'
```

(Observe the space before AND!) Now, if you go and add a lot of these SQL fragments, you will get them all on a single line, which may not be that readable. At the same time, putting the closing single quote on the next line would look funny. The way I usually deal with this is that I introduce a variable that I typically call **@nl** (for newline):

```
DECLARE @nl nchar(2) = char(13) + char(10)
```

And then I write the above as

```
IF @orderid IS NOT NULL
   SET @sql += ' AND O.OrderID = @orderid' + @nl
```

This gives me line breaks in my dynamic SQL. This little tip may not register with you right away, but I have found this approach to be very valuable to help me to write code that generates dynamic SQL, so that both the code itself and the generated code is clearly readable.

5.5 Dealing with Nested Strings

One of the most charming (irony intended) things when working with dynamic SQL is nested string literals. To include a single quote in a string in T-SQL you need to double it. You have already learnt that when it comes to strings where the values come from variables and where the syntax does not permit you to use the variables directly, you should use **quotename** or **quotestring** to get the value into the SQL string to protect yourself against SQL injection. This has the side effect that the amount of nested quotes is reduced.

To make it clear what I'm talking about, this is not the right way to do it:

```
CREATE OR ALTER PROCEDURE create_db @dbname   sysname,
                                    @sizeinmb smallint = 1200,
                                    @logsize  smallint = 200,
                                    @debug    bit = 0 AS

   DECLARE @datadir nvarchar(128) =
                  convert(nvarchar(128), serverproperty('InstanceDefaultDataPath')),
           @logdir nvarchar(128) =
                  convert(nvarchar(128), serverproperty('InstanceDefaultLogPath')),
           @sql   nvarchar(MAX)
   SELECT @sql =
     N'CREATE DATABASE ' + quotename(@dbname) + '
          ON PRIMARY (NAME = N''' + @dbname + ''',
             FILENAME = N''' + @datadir + @dbname + '.mdf'',
             SIZE = ' + convert(varchar, @sizeinmb) + ' MB)
          LOG ON (NAME = N''' +  @dbname + '_log'',
             FILENAME = N''' + @logdir + @dbname + '.ldf'',
             SIZE = ' + convert(varchar, @logsize) + ' MB)'
   IF @debug = 1 PRINT @sql
   EXEC(@sql)
```

As a reminder: the above will never run correctly on SQL 2008 or earlier as the two parameters to **serverproperty** were added in SQL 2012.

This code will fail if there is a single quote in the database name or in any of the installation paths. And it is open to SQL injection which would matter if **create_db** was exposed to plain users to permit them to create a database in a controlled way. (The procedure would somehow have to run with higher permissions that the users have themselves.) Add to this that is not trivial to write. It took me some attempts to get all the quotes rights, not the least all those Ns. As a repetition, here is the proper way to do it:

```
CREATE OR ALTER PROCEDURE create_db @dbname   sysname,
                                    @sizeinmb smallint = 1200,
                                    @logsize  smallint = 200,
                                    @debug    bit = 0 AS

   DECLARE @datadir nvarchar(128) =
                  convert(nvarchar(128), serverproperty('InstanceDefaultDataPath')),
           @logdir nvarchar(128) =
                  convert(nvarchar(128), serverproperty('InstanceDefaultLogPath')),
           @sql   nvarchar(MAX)
   SELECT @sql =
     N'CREATE DATABASE ' + quotename(@dbname) + '
          ON PRIMARY (NAME = N' + quotename(@dbname, '''') + ',
             FILENAME = N' + dbo.quotestring(@datadir + @dbname + '.mdf') + ',
             SIZE = ' + convert(varchar, @sizeinmb) + ' MB)
          LOG ON (NAME = N' +  quotename(@dbname + '_log', '''') + ',
             FILENAME = N' + dbo.quotestring(@logdir + @dbname + '.ldf') + ',
             SIZE = ' + convert(varchar, @logsize) + ' MB)'
   IF @debug = 1 PRINT @sql
   EXEC(@sql)
```

Here I have used **quotename** for the internal names of the files as they can at most be 128 characters, but **quotestring** for the file paths. But it would not be wrong to use **quotestring** for all four.

In terms of readability it may be a matter of taste which you prefer, but I suggest that the second one is easier to write. And moreover, it is safe for SQL injection.

We will look at some advanced techniques to deal with nested strings in the sections *More on Dynamic Names and Nested Strings* and *Playing with QUOTED_IDENTIFIER*.

Note: A better version of **create_db** would sanitise **@dbname** before constructing the file paths, so that it does not include any characters with special meaning to the file system, for instance slashes tilting forwards or backwards. I have left that out since it strays beyond the scope for this article.

5.6 A Trap with Long SQL Strings

Here is a something you can run into if you concatenate SQL strings that are quite long. Take a look at the script in the file longquery.sql. You find there a query which is seemingly non-sensical. In order to get a demo for this section, I took a long query from the MSDN forums, but to protect the poster, I replaced the original column names with product names from **NorthDynamic**. The original query did not use dynamic SQL, but I introduced a dynamic database name for the sake of the example for this section.

Note: to get the output described in this section, you need to have SQL 2012 or later. On SQL 2008, you will get additional errors due to the use of IIF.

The expected behaviour when running the script is that we will get an error message telling us that **View_XVUnitsProducts** is an invalid object, since there is no such view in **tempdb**. However, when we run the script, we get a different error:

```
Msg 102, Level 15, State 1, Line 49
Incorrect syntax near 'View_XVUnits'.
```

We can also see in the result set that **@sql** is 4000 characters long exactly, and if we look at the contents of **@sql**, we can see that the string is truncated. We are using **nvarchar(MAX)** for our **@sql** variable, and yet the query is truncated?

The underlying reason is found in the SQL Server type system. If you have an operation with two operands of the same data type, the result will be of the same data type as the operands. In this context, **nvarchar(20)** and **nvarchar(40)** can be considered the same data type, whereas **nvarchar(MAX)** is a different data type. In the script, the **@sql** variable is built up from a number of **nvarchar** literals interleaved with some calls to **quotename**. All string literals are less than 4000 characters in length. For this reason the type of the expression is **nvarchar(4000)**. The fact there is an **nvarchar(MAX)** variable on the left side of the assignment operator does not change this. Thus, the net result is that the string expression is silently truncated.

Note: for some extra spice, try replacing **tempdb** in the script with **NorthDynamic**. The way truncation strikes this time, the error message is definitely not easy to understand. At least as long as you do not look at the debug output.

At first, it may seem that to get this working, we have to wrap each string in **convert** or **cast** to change the data type to **nvarchar(MAX)**, which would make the code more difficult to read. Thankfully, it is not that bad. SQL Server has a rule which says that when two different types meet in the same expression, the type with lowest precedence in the SQL Server type system is converted to the type with higher precedence (if an implicit conversion exists, that is). For this reason, it is sufficient to introduce an initial **nvarchar(MAX)** value at the beginning of the concatenation, and this causes the conversion to snowball through the entire concatenation. Here is a possible solution (I'm only showing the beginning of the script):

```
DECLARE @sql nvarchar(MAX),
        @dbname sysname = 'tempdb'

SELECT @sql = cast('' AS nvarchar(MAX)) + N'
```

The error message is now the expected one:

```
Msg 208, Level 16, State 1, Line 2
Invalid object name 'tempdb.dbo.View_XVUnitsProducts'.
```

We can also see in the result set that the string is 4423 characters long. That is, by adding an empty string casted to **nvarchar(MAX)** we got the query working with minimal littering.

The **@sql** variable itself can serve for the task, and in many cases it comes naturally, because you add to the **@sql** string piece by piece. In this example it could be like this:

```
DECLARE @sql nvarchar(MAX) = '',
        @dbname sysname = 'tempdb'

SELECT @sql = @sql + N'
```

But there are two new traps hiding here. The first is one you would notice quickly. If you fail to initialise **@sql** to the empty string, the entire string will be NULL, and nothing would be executed. The other trap is that you (or someone else) may be tempted to introduce the shortcut operator +=:

```
DECLARE @sql nvarchar(MAX) = '',
        @dbname sysname = 'NorthDynamic'

SELECT @sql += N'
```

This will bring back the syntax error, because on the right-hand side of the += operator, there are now only short strings, and thus there is no implicit conversion to **nvarchar(MAX)**. For this reason, adding

```
cast('' AS nvarchar(MAX)) + N'
```

is a good safety precaution.

5.7 Receiving Names of Tables and Other Objects as Parameters

There are situations where you want to receive the name of a table or some other object as a parameter. Having table names as a parameter to a procedure in an application is not the best of ideas, something I will discuss further in the last chapter of this article. But there are certainly situations where this is can be desirable for a DBA task. Consider this procedure which may be not be the best example of a practical use case, but it serves as a short and concise example:

```
USE NorthDynamic
go
CREATE OR ALTER PROCEDURE GetCount @tblname nvarchar(1024) AS
   DECLARE  @sql nvarchar(MAX)
   SELECT @sql = 'SELECT COUNT(*) FROM ' + @tblname
   PRINT @sql
   EXEC sp_executesql @sql
go
EXEC GetCount 'dbo.Customers'
```

The example runs, but as you can see, the procedure is wide open to SQL injection. Nor is it able to handle names with special characters in them. We have learnt that we should use **quotename** to address this. But if we try:

```
CREATE OR ALTER PROCEDURE GetCount @tblname nvarchar(1024) AS
   DECLARE  @sql nvarchar(MAX)
   SELECT @sql = 'SELECT COUNT(*) FROM ' + quotename(@tblname)
   PRINT @sql
   EXEC sp_executesql @sql
go
EXEC GetCount 'dbo.Customers'
```

We get an error message:

```
SELECT COUNT(*) FROM [dbo.Customers]
Msg 208, Level 16, State 1, Line 7
Invalid object name 'dbo.Customers'.
```

Some readers may be puzzled by this, because at first glance it looks correct. Isn't there a table with this name in **NorthDynamic**? Nah, there is a table **Customers** in the **dbo** schema, and with brackets correctly applied, this is written as [**dbo**].[**Customers**]. But when there is only one pair of brackets, SQL Server takes **dbo.Customers** as a table name in one-part notation, and looks for it in the default schema of the current user.

That is, the dot is part of the name; it is not a separator when it is inside the brackets.

Here is another example where it goes wrong:

```
EXEC GetCount '[Order Details]'
```

The output:

```
SELECT COUNT(*) FROM [[Order Details]]]
Msg 208, Level 16, State 1, Line 10
Invalid object name '[Order Details]'.
```

This time it failed, because the caller already had wrapped the name in brackets.

The way to resolve this is to first use the **parsename** function to get any database part in **@tblname**. **parsename** is a function that returns the various parts of an object specification with the parts numbered from right to left. That is, 1 is the object name itself, 2 is the schema, 3 is the database name and 4 is the name of any linked server. If a part is missing from the specification, NULL is returned. Then use the **object_id** function to get the object id for the table/view and the function **db_id** to get the id of the database in **@tblname**, or the id of the current database, if no database component is present. Finally, you use the metadata function **db_name** to translate database id back to a name together with the functions **object_schema_name** and **object_name** to get the names of the schema and the object itself from the object and database ids. You wrap each component in **quotename**. Here is an example, with comments added to explain the various steps.

```
CREATE OR ALTER PROCEDURE GetCount @tblname nvarchar(1024) AS
   DECLARE  @sql       nvarchar(MAX),
            @object_id int,
            @db_id     int

   -- Supporting names on linked server would require a whole lot more work.
   IF parsename(@tblname, 4) IS NOT NULL
   BEGIN
      RAISERROR('Tables on a different server are not supported.', 16, 1)
      RETURN 1
   END

   -- Get object id and validate.
   SELECT @object_id = object_id(@tblname)
   IF @object_id IS NULL
   BEGIN
      RAISERROR('No such table/view "%s".', 16, 1, @tblname)
      RETURN 1
   END

   -- Get the database id for the object.
   IF parsename(@tblname, 3) IS NOT NULL
      SELECT @db_id = db_id(parsename(@tblname, 3))
   ELSE
      SELECT @db_id = db_id()

   -- Get the normalised name.
   SELECT @tblname = quotename(db_name(@db_id)) + '.' +
                     quotename(object_schema_name(@object_id, @db_id)) + '.' +
                     quotename(object_name(@object_id, @db_id))

   SELECT @sql = 'SELECT COUNT(*) FROM ' + @tblname
   PRINT @sql
   EXEC sp_executesql @sql
```

As the comment says, the procedure does not support objects on linked servers as that would be a whole lot more complicated, and I leave this as an exercise to the reader who needs this. This procedure supports objects in other databases, but in many cases you can permit yourself to constrain a lookup like this to the current database. In that case, you should add `parsename(@tblname, 3) IS NOT NULL` to the initial check and change the error message accordingly.

Here are some test cases:

```
CREATE DATABASE [extra-database]
go
USE [extra-database]
go
CREATE SCHEMA extra;
go
CREATE TABLE extra.testtable(a int NOT NULL)
INSERT extra.testtable(a) SELECT TOP 65 object_id FROM sys.objects
go
USE NorthDynamic
go
EXEC GetCount 'dbo.Customers'
EXEC GetCount '[Order Details]'
EXEC GetCount 'sys.schemas'
EXEC GetCount 'msdb.dbo.sysjobs'
EXEC GetCount '[extra-database].extra.testtable'
EXEC GetCount 'OTHERSERVER.master.sys.databases'
go
DROP DATABASE [extra-database]
```

When running the tests, we get this output in the *Messages* tab (with the "rows affected" messages omitted):

```
SELECT COUNT(*) FROM [NorthDynamic].[dbo].[Customers]
SELECT COUNT(*) FROM [NorthDynamic].[dbo].[Order Details]
SELECT COUNT(*) FROM [NorthDynamic].[sys].[schemas]
SELECT COUNT(*) FROM [msdb].[dbo].[sysjobs]
SELECT COUNT(*) FROM [extra-database].[extra].[testtable]
Msg 50000, Level 16, State 1, Procedure GetCount, Line 9 [Batch Start Line 50]
Tables on a different server is not supported.
```

You can apply the same pattern to other type of objects. For things like stored procedures and other things that live in **sys.objects** the example works as given (save for the SELECT COUNT(*) obviously). For other type of objects there are in many cases metadata functions you can use in the same manner; that is, you can translate to id and then back to a name you wrap in **quotename**. Alas, not all metadata functions understand quoted identifiers as testified by this example:

```
SELECT user_id('dbo'), user_id('[dbo]')   -- Returns 1, NULL
```

Thankfully, these are functions for classes of objects that do not live in a schema, and which neither can be addressed across databases, so the names always consist of a single component. This permits for an easier solution: check if the name is already wrapped in brackets, and in such case remove them before applying **quotename**. Here is an example with some test cases:

```
CREATE USER NormalUser WITHOUT LOGIN
CREATE USER AlsoNormal WITHOUT LOGIN
CREATE USER [Space user] WITHOUT LOGIN
go
CREATE OR ALTER PROCEDURE MyDropUser @username sysname AS
   DECLARE @sql nvarchar(MAX)

   SELECT @sql = 'DROP USER ' +
      quotename (CASE WHEN left(@username, 1) = '[' AND
                          right(@username, 1) = ']'
                     THEN substring(@username, 2, len(@username) - 2)
                     ELSE @username
                END)
   PRINT @sql
   EXEC(@sql)
go
EXEC MyDropUser 'NormalUser'
EXEC MyDropUser '[AlsoNormal]'
EXEC MyDropUser 'Space user'
```

This method does not handle exactly everything. For instance, if a user name actually starts and ends in brackets, things are likely to go wrong. But it may be good enough for you.

Some readers may think that if the name is wrapped in brackets already, we could take the string as-is without wrapping it in **quotename**, but that would be wrong. All say after me please: *SQL injection*!

5.8 Dynamic Database and Server Names

It is not uncommon that the database name is dynamic. It could be that you want to do something in a number of databases or because you have different databases for different years, customers or whatever. A variation of this theme is that in the same stored procedure you want to run commands both in the local database and in the **master** database. In this latter case, there is not really any dynamic with database name, but you need to switch databases.

Many people do as in this example:

```
DECLARE @dbname sysname = 'NorthDynamic',
        @sql    nvarchar(MAX)
SELECT @sql = 'USE ' + quotename(@dbname) + '
               SELECT * FROM dbo.Customers WHERE CustomerID = @custid'
PRINT @sql
EXEC sp_executesql @sql, N'@custid nchar(5)', N'ALFKI'
```

That works, but there is a different way that I find cleaner:

```
DECLARE @dbname        sysname = 'NorthDynamic',
        @sql           nvarchar(MAX),
        @sp_executesql nvarchar(200)
SELECT @sp_executesql = quotename(@dbname) + '.sys.sp_executesql'
SELECT @sql = N'SELECT * FROM dbo.Customers WHERE CustomerID = @custid'
PRINT @sql
EXEC @sp_executesql @sql, N'@custid nchar(5)', N'ALFKI'
```

I'm making use of two things here:

1. EXEC accepts a variable for the procedure name.
2. A system procedure executes in the context from the database it was invoked, also with three-part notation.

You could argue that an advantage with the USE statement is that the debug print makes it clear in which database the SQL batch is executed. That is certainly true, but you could address this by adding a comment to the SQL string.

We learnt in the section on EXEC() AT that we can use **sp_executesql** to run commands on a linked server that is another SQL Server instance. If you want the server name to be dynamic, you can use the same technique:

```
DECLARE @servername    sysname = 'YOURSERVER',
        @dbname        sysname = 'NorthDynamic',
        @sql           nvarchar(MAX),
        @sp_executesql nvarchar(200)
SELECT @sp_executesql = quotename(@servername) + '.' +
                        quotename(@dbname) + '.sys.sp_executesql'
SELECT @sql = N'SELECT * FROM dbo.Customers WHERE CustomerID = @custid'
PRINT @sql
EXEC @sp_executesql @sql, N'@custid nchar(5)', N'ALFKI'
```

Since USE is not easy to apply in this case, this solution comes out as really powerful here.

I should add that this technique is good when you want to run DDL commands or an occasional query in another database or server, but if you have plentiful of cross-database or cross-server queries in your application where you don't want to hardcode the database/server names, you should use synonyms instead, which relieves your from using dynamic SQL at all. I will discuss this further in the section *Setting up Synonyms for Cross-Database Access*.

5.9 Doing Something in All Databases

There are situations when you want to perform an action in all databases (or a selection of databases), and that action may include an operation on a suite of tables, views or whatever in the databases. There are two system procedures which are popular for this purpose, **sp_MSforeachdb** and **sp_MSforeachtable**. They are undocumented and unsupported, but you can easily find examples on the web.

Personally, I am not fond of these procedures, as the resulting code tends to be less than readable. If I want to do something in every database, I prefer to write my own cursor loop – that is after all not really rocket science. And if I want to perform the same action on a number of objects, I rather write a query against the system catalogue to generate the statements. If it is just a one-off, I simply copy the result into a query window and execute it. But if I need it in a script, I concatenate the rows into a single string and execute that string.

I will give you an example that you can use a boilerplate for your needs. The task is that we need to grant SELECT permissions on all views to a certain role in all databases were this role exists. Because we are only granting select on a certain object type, we cannot grant permission on schema or database level – we don't want to give the users access to the base tables – so we need to grant access per object. (Putting all views in a separate schema, so we could grant access on the schema? That's a great idea! But assume for the example that this is not feasible. Maybe we did not just think of it when we made our original design and now we are stuck with it.) Here is the script:

```
DECLARE @db            sysname,
        @sp_executesql nvarchar(500),
        @query         nvarchar(MAX),
        @dbstmts       nvarchar(MAX),
        @role          sysname = 'ourrole'

SELECT @query =
    'IF user_id(@role) IS NOT NULL
     BEGIN
        SELECT @dbstmts =
           (SELECT  ''GRANT SELECT ON '' +
                    quotename(s.name) + ''.'' + quotename(v.name) +
                    '' TO '' + quotename(@role) + char(13) + char(10)
            FROM    sys.views v
            JOIN    sys.schemas s ON s.schema_id = v.schema_id
            WHERE   NOT EXISTS (SELECT *
                                FROM   sys.database_permissions dp
                                WHERE  dp.major_id = v.object_id
                                  AND  dp.type = ''SL''
                                  AND  dp.grantee_principal_id = user_id(@role))
            FOR XML PATH(''''), TYPE).value(''.'', ''nvarchar(MAX)'')
     END
     ELSE
        SELECT @dbstmts = NULL'

DECLARE dbcur CURSOR STATIC LOCAL FOR
   SELECT quotename(name) FROM sys.databases
   WHERE  database_id > 4
     AND  state = 0          -- Only online databases
     AND  is_read_only = 0
   ORDER BY name

OPEN dbcur

WHILE 1 = 1
BEGIN
   FETCH dbcur INTO @db
   IF @@fetch_status <> 0
      BREAK

   SELECT @sp_executesql = @db + '.sys.sp_executesql'

   EXEC @sp_executesql @query, N'@dbstmts nvarchar(MAX) OUTPUT, @role sysname',
                       @dbstmts OUTPUT, @role
   PRINT @db
   IF @dbstmts IS NOT NULL
   BEGIN
      PRINT @dbstmts
```

```
         EXEC @sp_executesql @dbstmts
      END
   END

   DEALLOCATE dbcur
```

The script starts with a number of variables. **@db** is the database we are currently working with in the loop and **@sp_executesql** will hold the three-part name for **sp_executesql** in **@db**, as we discussed in the previous section. **@query** is the query we run in **@db** to generate **@dbstmts** which performs the actual work. All these four variables are part of the boilerplate, but the last **@role** is specific to this example: this is the role we will grant rights to. (I figured it was better to put the role name in a variable, in case we want to perform this action for more than one role.)

We first set up **@query**. It first checks if **@role** is present in this database (but it does not care that this is actually is a role, but any database principal will be accepted). If **@role** is present, **@query** runs the main query against **sys.views** to get the views, filtering out views that **@role** already have access to. The SELECT list builds a GRANT statement for each view; the statement is followed by CR+LF to make the output easy to read, in case we want inspect it.

You may not have seen this FOR XML PATH thing before: this is a way to convert a result set into a concatenated string. It works, but the syntax is not the prettiest. If you have no need to support SQL 2016 and earlier, you can instead use **string_agg** which is a lot cleaner. There is a version with **string_agg** below.

The next step is to set up a cursor over **sys.databases** so that we can loop over them. The cursor is STATIC LOCAL. It is entirely beyond the scope for this article, but I strongly recommend that you always set up your cursors this way, since else there can be nasty surprises. (The above-mentioned **sp_MSforeachdb** uses DYNAMIC GLOBAL, and there are reports of **sp_MSforeachdb** skipping databases.) Many times you will want to modify the filtering so that you include only databases of interest to you. In this example, I exclude the regular system databases, **master**, **tempdb**, **model** and **msdb**, but I make no effort to filter out databases for SSRS, replication etc. For this particular example, I trust that the check on the **@role** inside **@query** will filter out such databases anyway. But would you be doing something like creating a DDL trigger in all user databases, you really need to be careful so you don't create them in system-added databases.

I have two more filters: one to exclude databases that are not online, and one to exclude read-only databases. I would suggest that these are filters you will include more often than not. All depending on your needs you may want to add more conditions to exclude / include databases.

Inside the cursor loop it is quite straight-forward. We first execute **@query** to get the statements to execute in the current database in the loop. In many cases when you use this technique, **@query** will only have one parameter, that is, the output parameter **@dbstmts**. However, in this particular example we also send in **@role** as a parameter. It is a matter of taste, but I have opted to always print out the database name, even if there is nothing to execute for a particular database. Finally in the loop, we print **@dbstmts** and execute the SQL returned from **@query**.

Here is an alternate statement to build **@query** that uses the **string_agg** function which is prettier than the FOR XML PATH hocus pocus, but as noted it requires SQL 2017 or later:

```
   SELECT @query =
       'IF user_id(@role) IS NOT NULL
        BEGIN
           SELECT @dbstmts =  string_agg(convert(nvarchar(MAX),
                                 ''GRANT SELECT ON '') +
                                    quotename(s.name) + ''.'' + quotename(v.name) +
                                 '' TO '' + quotename(@role),
                                 nchar(13) + nchar(10))
               FROM    sys.views v
               JOIN    sys.schemas s ON s.schema_id = v.schema_id
               WHERE   NOT EXISTS (SELECT *
                                   FROM   sys.database_permissions dp
                                   WHERE  dp.major_id = v.object_id
                                     AND  dp.type = ''SL''
                                     AND  dp.grantee_principal_id = user_id(@role))
```

```
      END
      ELSE
         SELECT @dbstmts = NULL'
```

The **convert** to **nvarchar(MAX)** is needed to avoid truncation if the result is longer than 4000 characters.

You will see more of concatenation with FOR XML PATH and **string_agg** in later sections, and not the least in the section on dynamic pivot.

5.10 Cursors and Dynamic SQL

Cursors is something you should use sparingly, and the situations where you need to use dynamic SQL and a cursor in combinations are few and far between. Nevertheless, I have a tip to share, so here it goes.

I always tell people to set up their cursors this way:

```
DECLARE cur CURSOR STATIC LOCAL FOR
   SELECT ...
```

STATIC ensures that you don't get the default cursor type, which is DYNAMIC and which is prone to really bad performance. LOCAL means that the scope of the cursor is local to the stored procedure. That is, when the procedure exits, the cursor goes away. The default is that a cursor is global to your process. This can result in nasty errors if the cursor loop is interrupted half-way, and you re-enter the procedure on the same connection, as the cursor still exists at this point.

However, LOCAL does not work if you for whatever reason want to set up the cursor with dynamic SQL. If you try this:

```
DECLARE @name sysname,
        @sql  nvarchar(MAX)

SELECT @sql =
   'DECLARE cur CURSOR STATIC LOCAL FOR
       SELECT TOP (5) CompanyName FROM dbo.Customers
       ORDER  BY CustomerID'

EXEC sp_executesql @sql

OPEN cur
```

You get this error message for the OPEN statement:

```
Msg 16916, Level 16, State 1, Line 9
A cursor with the name 'cur' does not exist.
```

You should know why by now: the code inside **@sql** is a stored procedure of its own, and LOCAL means local to that scope.

But there is a way out that I originally learnt from Anthony Faull. You can use a cursor variable:

```
DECLARE @name sysname,
        @sql  nvarchar(MAX),
        @cur  CURSOR

SELECT @sql =
   'SET @cur = CURSOR STATIC FOR
       SELECT TOP (5) CompanyName FROM dbo.Customers
       ORDER  BY CustomerID
   OPEN  @cur'

EXEC sp_executesql @sql, N'@cur CURSOR OUTPUT', @cur OUTPUT

WHILE 1 = 1
BEGIN
   FETCH @cur INTO @name
   IF @@fetch_status <> 0
```

```
         BREAK

      PRINT @name
   END
```

Cursor variables have been in the product for a long time (since SQL 7), but they have remained one of those obscure features that few people know about and even less use. (Until Anthony sent me his example, I had never considered them myself.)

Pay attention to that the cursor is set up with SET rather than DECLARE. I still specify STATIC, but LOCAL is not needed (nor is it permitted), since the scope of the cursor is the same as for the variable that holds the cursor. Observe that the OPEN statement must be inside the dynamic SQL. If you put it after the call to dynamic SQL, you get this error:

```
   Msg 16950, Level 16, State 2, Line 13
   The variable '@cur' does not currently have a cursor allocated to it.
```

This is one of the very few cases where there is a difference between the nameless stored procedures created with **sp_executesql** and "real" stored procedures. With the latter, you need to insert the keyword VARYING between CURSOR and OUTPUT in the parameter declaration. But VARYING is not permitted with **sp_executesql**, and nor is it needed.

5.11 A Clean Way to Deal with Dynamic Names

We are starting to arrive at the more advanced sections of this chapter. Although the ideas in this section is something that everyone can have use for at times, inexperienced readers may prefer to jump to the next chapter at this point.

Dynamic SQL is difficult to read and maintain due to the mix of the syntax in the hosting procedure and the dynamic SQL. The sorest points are where you break up the SQL string to splice in a variable for something that cannot be a parameter. For instance, we previously looked at the procedure **create_db** which contains this statement:

```
      SELECT @sql =
        N'CREATE DATABASE ' + quotename(@dbname) + '
             ON PRIMARY (NAME = N' + quotename(@dbname, '''') + ',
                FILENAME = N' + dbo.quotestring(@datadir + @dbname + '.mdf') + ',
                SIZE = ' + convert(varchar, @sizeinmb) + ' MB)
             LOG ON (NAME = N' +  quotename(@dbname + '_log', '''') + ',
                FILENAME = N' + dbo.quotestring(@logdir + @dbname + '.ldf') + ',
                SIZE = ' + convert(varchar, @logsize) + ' MB)'
```

But you can avoid all these single quotes, plusses and function calls. Here is a version of **@sql** which tells what you want to do much more succinctly:

```
      SELECT @sql =
        N'CREATE DATABASE @dbname
             ON PRIMARY (NAME = @datafilename, FILENAME = @datafilepath,
                         SIZE = @sizeinmb MB)
             LOG ON (NAME = @logfilename, FILENAME = @logfilepath,
                     SIZE = @logsize MB)'
```

No, this is not legal syntax and these things starting with @ are not really variables, but rather they are placeholders which you replace with the real values with help of the **replace** function. Here is the full code of the new version of **create_db** together with a test case:

```
   CREATE OR ALTER PROCEDURE create_db @dbname   sysname,
                                       @sizeinmb smallint = 1200,
                                       @logsize  smallint = 200,
                                       @debug    bit = 0 AS

      DECLARE @datadir nvarchar(128) =
                 convert(nvarchar(128), serverproperty('InstanceDefaultDataPath')),
              @logdir nvarchar(128) =
                 convert(nvarchar(128), serverproperty('InstanceDefaultLogPath')),
              @sql   nvarchar(MAX)

      SELECT @sql =
        N'CREATE DATABASE @dbname
             ON PRIMARY (NAME = @datafilename, FILENAME = @datafilepath,
                         SIZE = @sizeinmb MB)
             LOG ON (NAME = @logfilename, FILENAME = @logfilepath,
                 SIZE = @logsize MB)'

      SELECT @sql = replace(@sql, '@dbname', quotename(@dbname))
      SELECT @sql = replace(@sql, '@datafilename',
                            'N' + quotename(@dbname, ''''))
      SELECT @sql = replace(@sql, '@datafilepath',
                            dbo.quotestring_n(@datadir + @dbname + '.mdf'))
      SELECT @sql = replace(@sql, '@sizeinmb', convert(varchar, @sizeinmb))
      SELECT @sql = replace(@sql, '@logfilename',
                            'N' + quotename(@dbname + '_log', ''''))
      SELECT @sql = replace(@sql, '@logfilepath',
                            dbo.quotestring_n(@logdir + @dbname + '.ldf'))
      SELECT @sql = replace(@sql, '@logsize', convert(varchar, @logsize))

      IF @debug = 1 PRINT @sql
      EXEC(@sql)
   go
   EXEC create_db [Test'Dβ], @debug = 1
   go
   DROP DATABASE [Test'Dβ]
```

Each **replace** also makes sure that the value is appropriately quoted. You may note that **@sql** has different variables/placeholders for the database name and the name of the primary data file, although the value is the same. This is required, since the value is to be quoted differently in the two places. You may also note that in this example, I use **quotestring_n** for the file paths, since they potentially can be more than 128 characters long. Note also that there is some importance to have the N before the string literals in all places, or else the Greek letter β will be replaced by its look-alike, the German "scharfes-s", ß.

This example may not be entirely persuasive. Although the **@sql** string as such is more concise and easier to understand, the total length of **create_db** more than doubled, so I would expect many readers to prefer the previous version. But imagine a longer SQL string that runs over twenty lines or more, with a just a couple unique "variables" that are repeated several times in the string. Now this technique is becoming more attractive. My thinking is that this is a trick to have your toolbox, when you feel that the string concatenation is threating your mental sanity, but this is nothing you would use in every case when you work with dynamic names. Use what you think works best for the situation.

We will explore more possibilities in the next section.

5.12 More on Dynamic Names and Nested Strings

In this section we will take a look at one more approach to deal with dynamic names, and we will also look some more techniques to deal with nested strings. What I present in this section is more directed to that advanced users with a good understanding of what they are doing, and less experienced readers may find the material a little too bewildering.

If we look at the solution for **create_db** in the previous section, it is a little frustrating that we have so many variables/placeholders that are derived from the input parameter **@dbname**. But rather than replacing the full values and adding string delimiters and all when we use **replace**, we could have placeholders inside the string literals, something like this:

```
        SELECT @sql =
          N'CREATE DATABASE @dbname
               ON PRIMARY (NAME = N''%(dbname)'',
                           FILENAME = N''%(datadir)%(dbname).mdf'',
                           SIZE = %(sizeinmb) MB)
               LOG ON (NAME = N''%(dbname)_log'',
                       FILENAME = N''%(logdir)%(dbname).ldf'',
                       SIZE = %(logsize) MB)'
```

If you are saying that you like the previous solution better, I am not going argue with you, but pretend for a second that you actually like this and hang on, because you may appreciate some of the devices I introduce to deal with nested strings. As for the style of the `%(placeholder)`, that's just something picked to have instead `@placeholder`, but you use what you prefer.

When we replace the placeholders, we must consider the risk that the values include single quotes, so it has to be something like this:

```
SELECT @sql = replace(@sql, '%(dbname)', replace(@dbname, '''', ''''''''))
```

Ouch! That was not pretty. Thankfully, we don't have to write code like that, but we can introduce two local variables to preserve our sanity:

```
DECLARE @sq    nchar(1) = '''',
        @sqsq  nchar(2) = '''' + ''''
```

Another thing which is a little bit of strain on the eye is all the doubled single quotes in the SQL string. This example may be somewhat tolerable, but if your dynamic SQL has a lot of constant string literals, you can really long for an alternative. What do you think about this:

```
   SELECT @sql =
     N'CREATE DATABASE @dbname
          ON PRIMARY (NAME = N"%(dbname)",
                      FILENAME = N"%(datadir)%(dbname).mdf",
                      SIZE = %(sizeinmb) MB)
          LOG ON (NAME = N"%(dbname)_log",
                  FILENAME = N"%(logdir)%(dbname).ldf",
                  SIZE = %(logsize) MB)'
   SELECT @sql = replace(@sql, '"', @sq)
```

That is, here I use double quotes, and then I replace the double quotes with single quotes once I have built my dynamic SQL.

While this can be more lenient on the eyes, this is a technique you must use with some care. You cannot do this, if you somewhere concatenate values into the string rather than using placeholders. Consider for the sake of the example, this mix:

```
SELECT @sql =
  N'CREATE DATABASE @dbname
       ON PRIMARY (NAME = N"%(dbname)",
                   FILENAME = N' + dbo.quotestring(@datadir + @dbname + '.mdf') + ',
                   SIZE = %(sizeinmb) MB)
       LOG ON (NAME = N"%(dbname)_log",
               FILENAME = N"%(logdir)%(dbname).ldf",
               SIZE = %(logsize) MB)'
  SELECT @sql = replace(@sql, '"', @sq)
```

If **@datadir** or **@dbname** would include a double quote you could get unexpected results, including syntax errors. And your code would be open to SQL injection.

From this follows that if you use double quote as an alternative string delimiter, you must also use variables/placeholders that you replace for *all* input values. And also very important: you must replace the double quotes with single quotes before you tackle any of the placeholders.

It goes without saying that if the SQL string includes double quotes naturally (e.g., XML, full-text predicates), you cannot use this technique with less than you use a different delimiter.

Adding all these pieces together, here is a new version of **create_db**:

```
CREATE OR ALTER PROCEDURE create_db @dbname   sysname,
                                    @sizeinmb smallint = 1200,
                                    @logsize  smallint = 200,
                                    @debug    bit = 0 AS
```

```
   DECLARE @datadir nvarchar(128) =
                convert(nvarchar(128), serverproperty('InstanceDefaultDataPath')),
           @logdir nvarchar(128) =
                convert(nvarchar(128), serverproperty('InstanceDefaultLogPath')),
           @sql   nvarchar(MAX),
           @sq    nchar(1) = '''',
           @sqsq  nchar(2) = '''' + ''''

   SELECT @sql =
     N'CREATE DATABASE @dbname
          ON PRIMARY (NAME = N"%(dbname)",
                      FILENAME = N"%(datadir)%(dbname).mdf",
                      SIZE = %(sizeinmb) MB)
          LOG ON (NAME = N"%(dbname)_log",
                  FILENAME = N"%(logdir)%(dbname).ldf",
                  SIZE = %(logsize) MB)'

   SELECT @sql = replace(@sql, '"', @sq)
   SELECT @sql = replace(@sql, '@dbname',     quotename(@dbname))
   SELECT @sql = replace(@sql, '%(dbname)',   replace(@dbname, @sq, @sqsq))
   SELECT @sql = replace(@sql, '%(datadir)',  replace(@datadir, @sq, @sqsq))
   SELECT @sql = replace(@sql, '%(logdir)',   replace(@logdir, @sq, @sqsq))
   SELECT @sql = replace(@sql, '%(sizeinmb)', convert(varchar, @sizeinmb))
   SELECT @sql = replace(@sql, '%(logsize)',  convert(varchar, @logsize))

   IF @debug = 1 PRINT @sql
   EXEC(@sql)
go
EXEC create_db [Test'Dβ], @debug = 1
go
DROP DATABASE [Test'Dβ]
```

5.13 Playing with QUOTED_IDENTIFIER

There is one more approach to nested strings which I mention with some hesitation, since it relies on deprecated functionality. Below is an excerpt from a script to collect a lot of data for all user databases on a server. (This is a script that I send to customers that want me to review the performance of their application.)

```
CREATE TABLE #sql (sql nvarchar(MAX) NOT NULL)
go
SET QUOTED_IDENTIFIER OFF
go
INSERT #sql (sql)
VALUES ("
INSERT tempdb..procs
SELECT db_name(), o.type, s.name, o.name, o.object_id
FROM   sys.objects o
JOIN   sys.schemas s ON o.schema_id = s.schema_id
WHERE  o.type IN ('P', 'PC', 'FN', 'FS')
...

INSERT tempdb..qs_runtime_stats_interval
    SELECT db_name() AS dbname, * FROM sys.query_store_runtime_stats_interval
")
go
SET QUOTED_IDENTIFIER ON
go
DECLARE @sql nvarchar(MAX),
        @sp_executesql nvarchar(200)
SELECT @sql = sql FROM #sql

DECLARE cur CURSOR STATIC LOCAL FOR
   SELECT quotename(name) + '.sys.sp_executesql'
   FROM   sys.databases
   WHERE  database_id >= 5
     AND  state = 0
     AND  is_read_only = 0

OPEN cur

WHILE 1 = 1
```

```
   BEGIN
      FETCH cur INTO @sp_executesql
      IF @@fetch_status <> 0
         BREAK

      EXEC @sp_executesql @sql
   END
```

I make use of that with SET QUOTED_IDENTIFIER OFF, the double quote now serves as string delimiter on equal footing with the single quote. That permits me to delimit the SQL string in double quotes, and there is no need to double all the single quotes. Which means that I can easily copy the SQL code to a query window in SSMS and test. (There is no dynamic part in the SQL code here; the only thing that requires dynamic SQL is that I want to run it in multiple databases.)

You should be aware of that SET QUOTED_IDENTIFIER OFF is a legacy option, and there are several features in SQL Server that require this setting to be ON, for instance XML type methods and filtered indexes to name two. To avoid any such accidents, I have isolated this setting to a single batch where I insert the SQL string into a temp table. And the reason for the temp table is exactly to permit me to have this isolated batch, else I could have assigned the text to **@sql** directly.

It goes without saying, that this is not going to work out well, if your SQL batch includes double quotes already, for instance because you are dealing with XML or JSON.

5.14 Further Examples

On my web site there are a couple of examples of what I think is well-crafted dynamic SQL. The examples are too long to include here, not the least, because it would take up too much space to explain the purpose of the code. So I let it suffice just to give the links to explore this code on your own, if you want inspiration for how to write good dynamic SQL.

Certificate signing

I have already referred to my article *Packaging Permissions in Stored Procedures* a few times in this article. This article includes a stored procedure **GrantPermsToSP** and a script GrantPermsToSP_server.sql which both automate the process of creating a certificate, signing a procedure, creating a user/login from that certificate and granting that user the permissions needed and thereby packaging the permissions into the procedure. The server-level script also copies the certificate from the user database to **master**. I think both of them are good examples of dynamic SQL, and you will recognise some of the techniques I have presented in this chapter.

The stored procedure is surely easier to digest, since it operates inside a single database. The server-side script is more complex. It also includes one thing I do not discuss elsewhere in this article: how to perform an action on all servers in an availability group.

beta_lockinfo

beta_lockinfo is a monitoring procedure that returns information about the current activity in the server: You can see the locks, who is blocking whom, the current statement of each process with the query plan and a lot more information. There is no end of dynamic SQL in this procedure. The longest piece is the place where I translate the various ids in **sys.dm_tran_locks** to names, which could be table names, index names etc. Here I make use of some the techniques I have discussed earlier in this style guide. There is also some more esoteric stuff. I wrote the first version of this procedure for SQL 6.5, and in those days there were no grids in the query tool, but all output was text-based. To optimize the screen estate, I added code to make the width of the columns to be dynamic (a technique I stole from **sp_who2**). It is still there if you follow the **@textmode** parameter.

For a maximum dosage of dynamic SQL, I think the version for SQL 2012-2014 is the best. (Because that version includes a part that only runs on SQL 2014, which I handled with dynamic SQL rather than making a new version of the procedure.)

6. Some Good Use Cases for Dynamic SQL

In this chapter we will look at some examples where dynamic SQL comes in handy. Some of the sections in this chapter are just a quick discussion, maybe with pointers elsewhere, whereas other includes actual solutions on how to do things. If you feel that your favourite use case is missing, drop me a line – but first read the next chapter where I discuss situations where dynamic SQL is less likely to be the right solution in my opinion. I mean, just in case your suggestion would be covered there. :-)

6.1 Dynamic Search Conditions

A requirement that comes up in more than one application is that users to want to be able to search the data using a lot of different search conditions. For instance, in an order system, users may want to find orders by order id, customer name, date interval, product id etc and often in combination. For a data set of any size, it is impossible to get good performance unless you observe that you need different query plans for different search conditions. This can be achieved with a well-written query in static SQL, provided that you add the query hint OPTION (RECOMPILE) at the end of the query. However, these solutions tend to break down, when the conditions are more complex in nature. Also, if searches are very frequent, the recompilation can become a performance issue in itself. The alternative is to build the search query with dynamic SQL, where it is easier to add more complex conditions, and since each combination of search conditions gets its own cached plan, this reduces the amount of compilation considerably.

I have a separate article on my web site entitled *Dynamic Search Conditions* where I discuss how to implement searches with static SQL + OPTION (RECOMPILE) as well as how to implement such searches with dynamic SQL, and thus I will not cover this topic more in this article.

6.2 Accessing Remote Data Sources

We have already looked a little bit at using dynamic SQL with linked servers, and particularly the case where you want the name of linked server to be set at run-time. In this section, we will work with cases where the name of the linked server is static. Furthermore, we will only look at queries that as such can be written with fully static SQL. But where we may still need to resort to dynamic SQL in order to get acceptable performance.

The simplest way to get data from a remote server is to refer to tables by four-part notation, that is **server.database.schemaname.tablename**. Sometimes this works just fine, but there are also situations when the optimizer decides for one reason or another to drag a big remote table in its entirety over to the local server to apply a WHERE condition which filters out all but a handful of rows.

When this happens, you start to look at alternatives. Closest at hand is OPENQUERY which permits you run a pass-through query on the remote data source, and thus ensures that filtering will be performed remotely. This often sends you to dynamic-SQL land, because OPENQUERY requires a fixed SQL string as input. You cannot pass the query text in a variable, and nor can the query be parameterised; all parameter values must be inlined to the query. (Exactly what I've been nagging in this article that you should never do!) This can lead to some quite ugly encounters with dynamic SQL.

There is a good reason why OPENQUERY does not accept a variable for the query: SQL Server needs to know the shape and format of the result set coming back from the query at compile time. And since the plan is cached, the result set must be known and fixed. Thus, the query cannot be a variable which produces results with different shapes on different executions. As for why OPENQUERY does not accept parameters, I am less sure, but it could be that they have simply not come around to implement it.

The best solution is to avoid OPENQUERY altogether, and instead run a dynamic parameterised query and receive the result into a temp table, which you can use in the rest of the query. If the remote server is another SQL Server instance, this is a pattern we have seen a few times already.

```
INSERT #temp (....)
   EXEC SERVER.db.sys.sp_executesql @query, @params, @par1, @par2, ...
```

This will execute on the remote server in the context of the remote database, so in **@query** you can refer to tables in two-part notation; no need to specify the database name in the query. If the linked server is something else than SQL Server, you cannot use **sp_executesql**, but earlier in the article we looked at EXEC() AT which also supports parameters.

There are two reasons why this approach is to prefer over OPENQUERY:

- The dynamic SQL is less ugly – typically **@query** is a fixed query string.
- You avoid littering the plan cache on the remote server by sending different query strings for different parameter values. (This applies if the remote server is another SQL Server instance. It may or may not apply to other remote data sources as well.)

Nevertheless, there may be situations where this solution is not to your liking and you find that OPENQUERY works best for you, so let's look at how do this in a proper way. Before we go on, we need a linked server to have something to play with. We set up a loopback server in the section on EXEC() AT, but in case you don't have it around anymore here are the commands again:

```
EXEC sp_addlinkedserver LOOPBACK, '', 'SQLNCLI', @datasrc = @@servername
EXEC sp_serveroption LOOPBACK, 'rpc out', 'true'
```

As I said before, a loopback server like this permits us to run the test scripts from a single query window. If you prefer, you can use a second instance as your linked server. Just make sure that you have **NorthDynamic** on that instance.

If you start using OPENQUERY without too much thought behind it, you may end up writing something like this:

```
CREATE OR ALTER PROCEDURE get_remote_data @searchstr nvarchar(40) AS
DECLARE @sql nvarchar(MAX)
SELECT @sql = 'SELECT * FROM OPENQUERY(LOOPBACK, ' +
              '''SELECT * FROM NorthDynamic.dbo.Customers ' +
              'WHERE CompanyName LIKE N''''%' + @searchstr + '%'''''')'
PRINT @sql
EXEC sp_executesql @sql
go
EXEC get_remote_data 'Futter'
```

I can only recommend you to take this path, if you already have made a conscious decision to lose your mind. Do you really fancy the idea of staring at something like this for a whole day, trying to understand if the reason why it does not work is because you have one single quote too many or too few somewhere? Or whether that per-cent characters should be elsewhere? The above does work – but I can assure you that I did not get it right in my first attempt. And while it may be working, this is open for SQL injection – both on the remote server and the local server.

When you work with OPENQUERY, you need to take a systematic approach when you build your dynamic query and do it in two steps. First build the remote query, and then embed it into the local query, as in this example:

```
CREATE OR ALTER PROCEDURE get_remote_data @searchstr nvarchar(40) AS
DECLARE @remotesql nvarchar(MAX),
        @localsql  nvarchar(MAX)

SELECT @remotesql =
     'SELECT * FROM NorthDynamic.dbo.Customers
      WHERE CompanyName LIKE ' + dbo.quotestring_n('%' + @searchstr + '%')
SELECT @localsql = 'SELECT * FROM OPENQUERY(LOOPBACK, ' +
                   dbo.quotestring_n(@remotesql) + ')'
PRINT @localsql
EXEC sp_executesql @localsql
go
EXEC get_remote_data 'snabb'
```

Note that we must use **quotestring_n** here for **@remotesql**, since most likely it will exceed 128 characters, so **quotename** is not a viable option. We could use **quotename** for **@searchstr** since it is at most 40 characters, but it seems logical to use the same for both. As whether to use **quotestring** or **quotestring_n**, that's not only a matter of being pretty. The string literal passed to OPENQUERY must be **nvarchar**, or else **@searchstr** could be used for SQL injection on the remote server like the trick Joe Cool played in the section *The Importance of nvarchar*.

We will now move on to another situation where performance can be a challenge.

```
SELECT ...
FROM   server.db.dbo.bigremotetbl r
JOIN   dbo.tinylocaltbl l ON r.somecol = l.somecol
```

As the names suggest, **bigremotetbl** is a big table, with millions of rows or more, while **tinylocaltbl** has just a handful. For good performance you want the optimizer to settle for a plan where the contents of **tinylocaltbl.somecol** is sent over to the remote server, and the part of the query against **bigremotetbl** is evaluated there. The optimizer is certainly able to produce such a plan, but for one reason or another, it could get the idea of bringing over all rows in **bigremotetbl** to the local server. I will have to admit that I am not inclined to take my chances that it will get it right. When I encounter a situation like this, I rather pass the data in the local table to the remote server and run a query there as a disaster prevention.

My preferred method is to package the data in the local table in an XML string which I pass to the remote server with **sp_executesql**. In the remote query, I shred the XML data into a local temp table which I then join to the main table on the remote server. Here is an example where the temp table **#getthese** corresponds to **tinylocaltbl** while **Customers** in **NorthDynamic** serves as a token **bigremotetbl**.

```
CREATE TABLE #getthese (custid nchar(5) NOT NULL PRIMARY KEY)
INSERT #getthese(custid)
   VALUES(N'ALFKI'), (N'BERGS'), (N'VINET')

go
DECLARE @xmlstr nvarchar(MAX),
        @query  nvarchar(MAX)
SELECT @xmlstr = (SELECT custid FROM #getthese FOR XML RAW, ROOT('root'))

SELECT @query = '
   DECLARE @xml xml = cast(@xmlstr AS xml)

   CREATE TABLE #xmldata (custid nchar(5) NOT NULL PRIMARY KEY)

   INSERT #xmldata(custid)
      SELECT T.c.value(''@custid'', ''nchar(5)'')
      FROM   @xml.nodes(''/root/row'') AS T(c)

   SELECT C.CustomerID, C.CompanyName, C.ContactName, C.City
   FROM   dbo.Customers C
   WHERE  EXISTS (SELECT * FROM #xmldata x WHERE C.CustomerID = x.custid)'

EXEC LOOPBACK.NorthDynamic.sys.sp_executesql @query, N'@xmlstr nvarchar(MAX)', @xmlstr
```

When forming the XML string, I use FOR XML RAW, which is the easiest to use for this purpose. You may note that the data type of **@xmlstr** is **nvarchar** and not **xml**. This is because, as we have noted a few times already, the **xml** data type is not supported in calls to linked servers. Instead, I cast **@xmlstr** to **xml** on the remote side, so that I can shred it into **#xmldata**.

You may ask if that temp table on remote server is needed, couldn't we use **@xml.nodes** directly? Indeed, we can, but then optimizer would not have any clue about the data, but make a blind assumption which could lead to poor performance. For this reason, it is generally a good idea to shred XML data into a temp table, to give the optimizer some statistics to work from.

The reader may find this overly complicated and ask if we could not use a table-valued parameter instead. The answer is the same as for the **xml** data type: not supported with linked servers. What is a viable alternative on SQL 2016 and later is to use JSON instead if you so fancy. Since JSON is handled as **nvarchar** in SQL Server anyway, this removes the initial cast in **@query** above.

An alternative is to use OPENQUERY instead and transform the local temp table to an IN list:

```
DECLARE @getlist      nvarchar(MAX),
        @remotequery  nvarchar(MAX),
        @localquery   nvarchar(MAX)

SELECT @getlist =
   (SELECT dbo.quotestring_n(custid) + ', '
    FROM   #getthese
    FOR XML PATH(''), TYPE).value('.', 'nvarchar(MAX)')
SELECT @getlist = substring(@getlist, 1, len(@getlist) - 1)

-- SELECT @getlist = string_agg(dbo.quotestring_n(custid), ',') FROM #getthese

SELECT @remotequery =
    N'SELECT C.CustomerID, C.CompanyName, C.ContactName, C.City
      FROM   NorthDynamic.dbo.Customers C
      WHERE  C.CustomerID IN (' + @getlist + ')'

SELECT @localquery =
   N'SELECT CustomerID, CompanyName, ContactName, City
     FROM   OPENQUERY(LOOPBACK, ' + dbo.quotestring_n(@remotequery) + ')'

PRINT @localquery
EXEC sp_executesql @localquery
```

The comma-separated list is saved into **@getlist**, which we construct with help of FOR XML PATH. No, we are not really using XML here, this is just this funny way to build a string from a result set which have seen a few times already in this article. On SQL 2017 and later, we can use the more straightforward **string_agg** instead, here commented out. Once we have **@getlist**, we build the query in two steps as we did above: we first build **@remotequery**, and then we embed it into **@localquery**.

I am less fond of this solution. It causes litter in the plan cache on the remote server, since the values in **#getlist** appear as constants in the query. Furthermore, the IN operator is only useful when there is a single key column. With EXISTS you can also handle multi-key conditions. Then again, the solution with OPENQUERY should work with about any remote data source, whereas the solution with sending over XML would have to be reworked if your linked server is something else than SQL Server.

6.3 Setting up Synonyms for Cross-Database Access

You may have a situation where two databases on a server need to reference each other for one reason or another. If you hardcode references with three-part notation like **otherdb.dbo.tbl**, you get a problem if you want to set up a second environment for test or whatever on the same instance. If all cross-references are by procedure calls, you can build procedure names dynamically:

```
SELECT @otherdb = otherdb FROM configtable
SELECT @spname = @otherdb + '.dbo.some_sp'
EXEC @spname
```

But that is less palatable if you also want to make references to views or tables in the other database, since you would need to use dynamic SQL to get the database name into the query. Thankfully, there is no need to do this, since you can use synonyms. For instance:

```
CREATE SYNONYM otherdb.thattable FOR MyOtherDB.dbo.thattable
```

You can now query **otherdb.thattable** as if it were a local table, but the query will actually access **thattable** in **MyOtherDB**. In this example I have placed the synonym in a schema which is specific for references to the other database. There is no requirement to do this, but this works too:

```
CREATE SYNONYM dbo.thattable FOR MyOtherDB.dbo.thattable
```

However, having a separate schema helps you to see what the name refers to. The schema also makes it easier to administrate the synonyms with the procedures I present in this section. To wit, if you have a lot of objects you need to make references to, you need a lot of synonyms, and maintaining this manually is not really practical. This is a place where dynamic SQL comes in handy, and I will present two stored procedures, **setup_synonyms** and **retarget_synonyms**.

setup_synonyms sets up synonyms for all tables, views and stored procedures in **@otherdb** and puts the synonyms in the schema **@synschema**. The list of object types is hardcoded, but you can modify the list to fit your needs. All existing synonyms in **@synschema** are dropped before the new ones are created, so you can use **setup_synonyms** for an initial setup as well as for a refresh when new objects have been added to the other database. You can also use it to re-target the synonyms to refer to a different database, for instance when you have restored a set of databases with new names on the same instance. The statements to drop and create synonyms are inside a transaction, so that if something goes wrong, everything is rolled back. Feel free to adapt the procedure to fit your needs.

```
CREATE OR ALTER PROCEDURE setup_synonyms @synschema sysname, @otherdb sysname AS
BEGIN TRY
   DECLARE @sp_executesql nvarchar(600),
           @dropsql       nvarchar(MAX),
           @createquery   nvarchar(MAX),
           @createsql     nvarchar(MAX)

   -- Validate input parameters.
   IF schema_id(@synschema) IS NULL
      RAISERROR('Schema "%s" does not exist.', 16, 1, @synschema)

   IF db_id(@otherdb) IS NULL
      RAISERROR('Database "%s" does not exist.', 16, 1, @otherdb)

   -- Set up reference for sp_executesql.
   SELECT @sp_executesql =  quotename(@otherdb) + '.sys.sp_executesql'

   -- Run a query to generate the SQL to drop existing synonyms.
   SELECT @dropsql =
      (SELECT 'DROP SYNONYM ' + quotename(@synschema) + '.' + quotename(name) +
              char(13) + char(10)
       FROM   sys.synonyms
       WHERE  schema_id = schema_id(@synschema)
       FOR XML PATH(''), TYPE).value('.', 'nvarchar(MAX)')

   -- Set up a query to create the new synonyms. This is dynamic SQL, since
   -- runs in the other database, of which the name is a parameter.
   SELECT @createquery = 'SELECT @sql =
                      (SELECT '' CREATE SYNONYM '' +
                              quotename(@schema) + ''.'' + quotename(o.name) +
                              '' FOR '' + quotename(db_name()) + ''.'' +
                              quotename(s.name) + ''.'' + quotename(o.name) +
                              char(13) + char(10)
                      FROM  sys.objects o
                      JOIN  sys.schemas s ON o.schema_id = s.schema_id
                      WHERE o.type IN (''P'', ''U'', ''V'')
                      FOR XML PATH(''''), TYPE).value(''.'', ''nvarchar(MAX)'')'

    -- Run query to get the statements.
    EXEC @sp_executesql @createquery, N'@schema sysname, @sql nvarchar(MAX) OUTPUT',
                                      @synschema, @createsql OUTPUT

    -- Print and run the statements inside a transaction
    BEGIN TRANSACTION
    PRINT @dropsql
    EXEC (@dropsql)
    PRINT @createsql
    EXEC (@createsql)
    COMMIT TRANSACTION
END TRY
BEGIN CATCH
    IF @@trancount > 0 ROLLBACK TRANSACTION
    ; THROW
END CATCH
```

Note: the CATCH handler uses the ;THROW statement that was introduced in SQL 2012. For a quick test, you can replace ;THROW with a RAISERROR statement, if you are on SQL 2008 and earlier. For more information about error handling, see Part One of my series *Error and Transaction Handling in SQL Server*.

There may be situations where you don't want to create synonyms for each and every object in the other database, for instance, because developers are only supposed to use officially defined interface objects. If so, you would define your synonyms in a script and not use **setup_synonyms**. However, you may still have the need to redefine all synonyms to refer to a different database, and to this end you would use **retarget_synonyms**. It updates all existing synonyms in **@synschema** by replacing the existing database name with **@otherdb**. As with **setup_synonyms**, all work is done inside a transaction.

```
CREATE OR ALTER PROCEDURE retarget_synonyms @synschema sysname, @otherdb sysname AS
BEGIN TRY
   DECLARE @sql  nvarchar(MAX)

   -- Validate input parameters.
   IF schema_id(@synschema) IS NULL
      RAISERROR('Schema "%s" does not exist.', 16, 1, @synschema)

   IF db_id(@otherdb) IS NULL
      RAISERROR('Database "%s" does not exist.', 16, 1, @otherdb)

   -- Run a query to generate the SQL to drop existing synonyms.
   SELECT @sql =
      (SELECT 'DROP SYNONYM ' + quotename(@synschema) + '.' + quotename(name) +
              char(13) + char(10) +
              'CREATE SYNONYM ' + quotename(@synschema) + '.' + quotename(name) +
                 ' FOR ' + quotename(@otherdb) + '.' +
                          quotename(parsename(base_object_name, 2)) + '.' +
                          quotename(parsename(base_object_name, 1)) +
              char(13) + char(10)
       FROM   sys.synonyms
       WHERE  schema_id = schema_id(@synschema)
       FOR XML PATH(''), TYPE).value('.', 'nvarchar(MAX)')

    -- Print and run the statements inside a transaction.
    BEGIN TRANSACTION
    PRINT @sql
    EXEC (@sql)
    COMMIT TRANSACTION
END TRY
BEGIN CATCH
    IF @@trancount > 0 ROLLBACK TRANSACTION
    ; THROW
END CATCH
```

I leave these procedures uncommented from the perspective of dynamic SQL, as they build on concepts we already have discussed, but you are welcome to study the principles that I employ. Like in previous examples where we have used FOR XML PATH to build a list of values, you can instead use **string_agg** on SQL 2017 or later to make the code simpler

The other database may exist on another SQL Server instance. You can use synonyms in this case as well like in this example:

```
CREATE SYNONYM otherdb.sometbl FOR SERVER.otherdb.dbo.sometbl
```

Modifying these procedures to support an optional **@server** parameter is fairly straightforward. You will need to change **@createquery** in **setup_synonyms** so that the **@sql** code is returned as a result set which you capture with INSERT-EXEC, since SQL Server does not support MAX types for OUTPUT parameters.

6.4 BULK INSERT and OPENROWSET(BULK)

If you want to load data from a file with BULK INSERT or OPENROWSET(BULK), you often find yourself using dynamic SQL, because they require that the names of all files in the command (data file, format file, error file) to be string literals, but very often the file name(s) are given at run-time. Since you are working with strings, the general caveats apply. That is, use one of **quotename** or **quotestring**. The latter is safer, since a file path certainly can be more than 128 characters long.

Here is a sample with BULK INSERT:

```
DECLARE @datafile nvarchar(250) = 'C:\temp\mydata.bcp',
        @formatfile nvarchar(250) = 'C:\temp\mydata.fmt',
        @sql nvarchar(MAX)

SELECT @sql = 'BULK INSERT mytable FROM ' + dbo.quotestring_n(@datafile) + '
               WITH (FORMATFILE = ' + dbo.quotestring_n(@formatfile) + ')'
PRINT @sql
EXEC(@sql)
```

The generated command is:

```
BULK INSERT mytable FROM N'C:\temp\mydata.bcp'
               WITH (FORMATFILE = N'C.\temp\mydata.fmt')
```

I'm not taking up space here with a sample table and sample files, so unless you create ones yourself, the command will fail with an error that the table or file is missing when you run it.

Here is an example with OPENROWSET(BULK):

```
DECLARE @datafile nvarchar(250) = 'C:\temp\mydata.bcp',
        @formatfile nvarchar(250) = 'C:\temp\mydata.fmt',
        @sql nvarchar(MAX)

SELECT @sql =
   'INSERT mytable (a, b, c)
      SELECT a, b, c FROM OPENROWSET(BULK ' + dbo.quotestring_n(@datafile) + ',
      FORMATFILE = ' + dbo.quotestring_n(@formatfile) + ') AS t'
PRINT @sql
EXEC(@sql)
```

One could argue that from the perspective of dynamic SQL, that it would be cleaner to have the INSERT outside of the dynamic SQL and use INSERT-EXEC. However, there are some optimisations when you say INSERT ... SELECT ... FROM OPENROWSET(BULK) which you lose if you put the INSERT outside the dynamic SQL. Likewise, there are some hints to the INSERT command that are only applicable to OPENROWSET(BULK). So for this reason, it is better to keep the INSERT inside the dynamic SQL.

6.5 Maintenance Tasks in General

There are a lot of DBA tasks that can be automated with help of dynamic SQL. Here are just some examples:

- Changing a lot of logins to enable/disable password policies.
- Creating logins and users from a table you have imported from an Excel file.
- Adding a DDL trigger to many databases to enforce some policy.
- Adding standard auditing columns to tables that do not have them.
- A script you run in maintenance windows to increase the size of data files that have less than 10 % of free space.

And there is no end to it. As long as you play the DBA and you are not writing application code, dynamic SQL is almost always a fair game. Just some caveats:

- Don't miss the section on BACKUP/RESTORE in the next chapter.
- If you are considering automating index and statistics maintenance, first visit http://ola.hallengren.com and look at his solution which is more or less the de-facto standard in the SQL Server world. Ola has already done the work for you.
- And while dynamic SQL in T-SQL can be a really good hammer, it is not always the right tool. Particularly, if you want to do something on many servers, you should really be using Powershell. And no matter if it is a task for a single server or many, it is always a good idea to check out whether the nice people at dbatools.io have something for you.

6.6 Dynamic Pivot – Making Rows into Columns

The upshot for this problem is that you have a dataset with two or more keys, for instance sales per product and employee. You want to present the data as a matrix with one column per employee. You don't want to hardcode the employees, but if a new employee joins the sales department, the output should automatically include that employee.

Let's get this clear from the start: this is a non-relational operation. A query in a relational database returns a table with a fixed number of columns, where each column represents a distinct attribute in the dataset being returned. A dynamic pivot is the antithesis of this. Or more to the point: it is a presentational device. And logically, the best place for such devices is in the presentation layer and you should expect a good report writer to provide such functionality. To mention a few, I know that in SSRS the Tablix report can do it, and I get the impression that in Power BI you can use Matrix Visual. Excel has had pivoting support for a long time. If you are receiving the data in, say, a .NET program, you can easily pivot the data in C#.

I could have stopped here and put this section into the next chapter as an example of what you should not use dynamic SQL for. However, there may still be situations where you prefer to implement the pivoting in T-SQL. For a very obvious example: you are a DBA, and your presentation layer is SSMS. In this case there is no other option. And even if you have a client program, you might find it more convenient to do the pivoting in T-SQL. Furthermore, I see questions almost daily in the SQL forums that I monitor that ask for a dynamic pivot. (Never mind that these questions are often from inexperienced users who in many cases would be better off exploring client-side options.) So for these reasons, I decided to write this section to show how to implement a dynamic pivot in a structured way.

Note: you may have been referred to this section directly in response to a forum question. If you are eager to find a solution to your problem at hand, you can go on reading. But be warned that I have written this section under the assumption that you have read at least chapters 2 and 3, so if you don't really grasp what I'm talking about, go to the top of the article, and come back here when you completed those chapters. If you want to play with the examples, you find information about the demo database **NorthDynamic** and applicable SQL versions in the short section *Applicable SQL Versions and Demo Database* in the beginning of the article.

As always when we work with dynamic SQL, we should first implement a static solution, so that we know the syntax of what we are going to generate. Let's say that we want to see how different products sell in the five most important countries for NorthDynamic Trading: Brazil, France, Germany, UK and USA. Some readers may at this point expect to see a solution using the PIVOT operator. However, I find this operator to be of little value. In fact, so little value, that I have not even bothered to learn it. Instead, we will use something I call *CASE filters* inside an aggregate function. Here is a query on this theme:

```
SELECT P.ProductName,
       SUM(CASE C.Country WHEN 'Brazil'  THEN OD.Amount END) AS Brazil,
       SUM(CASE C.Country WHEN 'France'  THEN OD.Amount END) AS France,
       SUM(CASE C.Country WHEN 'Germany' THEN OD.Amount END) AS Germany,
       SUM(CASE C.Country WHEN 'UK'      THEN OD.Amount END) AS UK,
       SUM(CASE C.Country WHEN 'USA'     THEN OD.Amount END) AS USA
FROM   Orders O
JOIN   [Order Details] OD ON O.OrderID = OD.OrderID
JOIN   Customers C ON C.CustomerID = O.CustomerID
JOIN   Products P ON P.ProductID = OD.ProductID
WHERE  C.Country IN ('Brazil', 'France', 'Germany', 'USA', 'UK')
GROUP  BY P.ProductName
ORDER  BY P.ProductName
```

As you see, we have fives calls to the SUM aggregate function, one for each country, and inside these we have a CASE expression with a single WHEN clause and no ELSE. The part about no ELSE is important. For SUM, it would work with adding `ELSE 0`, but that would not work with other aggregate functions because it would yield incorrect results. It is these single-alternative CASE expressions I refer to as CASE filters.

Note: if you don't immediately got what I meant with incorrect results, but want to know: Replace SUM with AVG and run the query with and without `ELSE 0` to see the difference. With `ELSE 0` in the query, all orders count, not only those from Brazil, France etc, so the averages are way too low.

Readers who are familiar with the PIVOT operator may wonder why I favour the above. True, it is certainly a little more verbose than a query using PIVOT. However, when we generate the query with dynamic SQL, this will remove the repetitive nature of the code, as you will see later. What is much more important is that the scheme above is very flexible. Say that we want two more columns: one for all other countries and a grand total per product. That is very simple, just remove the WHERE clause, and add one more filtered SUM and one unfiltered:

```
SELECT P.ProductName,
       SUM(CASE C.Country WHEN 'Brazil'  THEN OD.Amount END) AS Brazil,
       SUM(CASE C.Country WHEN 'France'  THEN OD.Amount END) AS France,
       SUM(CASE C.Country WHEN 'Germany' THEN OD.Amount END) AS Germany,
       SUM(CASE C.Country WHEN 'UK'      THEN OD.Amount END) AS UK,
       SUM(CASE C.Country WHEN 'USA'     THEN OD.Amount END) AS USA,
       SUM(CASE WHEN C.Country NOT IN ('Brazil', 'France', 'Germany', 'USA', 'UK')
                THEN OD.Amount
           END) AS Others,
       SUM(OD.Amount) AS [Grand Total]
FROM   Orders O
JOIN   [Order Details] OD ON O.OrderID = OD.OrderID
JOIN   Customers C ON C.CustomerID = O.CustomerID
JOIN   Products P ON P.ProductID = OD.ProductID
GROUP  BY GROUPING SETS (P.ProductName, ())
ORDER  BY GROUPING(P.ProductName), P.ProductName
```

Note: While I was at it, I also added a grand total per column with help of the GROUPING SETS keyword. However, since this has no relation to dynamic SQL, I leave this unexplained and refer you to Books Online if you want to know more. This syntax was introduced in SQL 2008 and does not run on SQL 2005.

Before we dive into to generating pivot queries dynamically, we will look at one more example. A pivot query always includes aggregate functions with CASE filters, and in most cases the aggregate comes in naturally, because you are looking for a sum, an average etc. But occasionally, you have a set of data with a two-column key that you want to present in matrix form without any aggregation. As an example we can take the view **ProductCountrySales** in **NorthDynamic**:

```
CREATE VIEW ProductCountrySales AS
   SELECT P.ProductName, C.Country, SUM(OD.Amount) AS TotAmount
   FROM   Orders O
   JOIN   [Order Details] OD ON O.OrderID = OD.OrderID
   JOIN   Customers C ON C.CustomerID = O.CustomerID
   JOIN   Products P ON P.ProductID = OD.ProductID
   GROUP  BY P.ProductName, C.Country
```

We like to use this view to return the same data as in the first example rather than using the base tables. But since the SUM is inside the view, there is no apparent need to perform any aggregation. Still, we need an aggregate function, because that is the only way to get data for the same product on a single line. The trick is to use MIN or MAX:

```
SELECT ProductName,
       MIN(CASE Country WHEN 'Brazil'  THEN TotAmount END) AS Brazil,
       MIN(CASE Country WHEN 'France'  THEN TotAmount END) AS France,
       MIN(CASE Country WHEN 'Germany' THEN TotAmount END) AS Germany,
       MIN(CASE Country WHEN 'UK'      THEN TotAmount END) AS UK,
       MIN(CASE Country WHEN 'USA'     THEN TotAmount END) AS USA
FROM   ProductCountrySales
WHERE  Country IN ('Brazil', 'France', 'Germany', 'USA', 'UK')
GROUP  BY ProductName
```

The reason this works is because the CASE filters make sure that every MIN (or MAX) only sees one single non-NULL value.

Let's now move over to generate pivot queries dynamically, so that we can get any number of columns in our pivot. A stored procedure that generates and runs a dynamic pivot typically consists of six steps:

1. Identifying the values that will define the columns in the query, and saving these into a temp table.
2. Creating the initial part of the query, that is SELECT + initial columns.

3. Generating all the required CASE-filtered calls to the aggregate function with help of the data in the temp table and adding these to the query string.
4. Adding any final columns in the desired result set to the query string.
5. Adding the rest of the query: FROM, JOIN, WHERE, GROUP BY and ORDER BY clauses.
6. Running the generated query.

We will go through these steps with the procedure **ProductCountrySales_sp**. The aim of this procedure is to return sales per product and country during a period, only including product and countries that actually had sales during this period. That is, if no customer from, say, Belgium, bought anything during the period, Belgium should not be included in the result set. The country columns should be sorted first alphabetically by continent, and then by country within the continent. In this first version, we do not include any subtotals per continent, but we will look at this at the end of the section.

We will go through this procedure step by step, although not in the order above, but rather the order you should follow when you implement a solution for a dynamic pivot. We start with an empty shell, with only step 6 included:

```
CREATE OR ALTER PROCEDURE ProductCountrySales_sp @fromdate date,
                                                 @todate   date,
                                                 @debug    bit = 0 AS
BEGIN TRY
   DECLARE @lineend char(3) = ',' + char(13) + char(10),
           @sql     nvarchar(MAX)

   -- More code will added as we go on.

   IF @debug = 1
      PRINT @sql
   EXEC sp_executesql @sql, N'@fromdate date, @todate date', @fromdate, @todate
END TRY
BEGIN CATCH
   IF @@trancount > 0 ROLLBACK TRANSACTION
   ; THROW
END CATCH
```

The procedure takes three parameters. Beside **@fromdate** and **@todate** dictated by the desired functionality, there is also a **@debug** parameter. As I have pointed out a few times, this parameter is mandatory when you work with dynamic SQL so that you can inspect the dynamic SQL if things go wrong. The last statement in the procedure is to run the dynamic SQL that we will learn to generate in the text that follows. We can see that this SQL string will have **@fromdate** and **@todate** as parameters; we will of course not embed the values of these parameters into the SQL string.

The logic of the procedure is wrapped in TRY-CATCH. This has nothing to do with dynamic SQL, but I'm including it for the sake of best practice. If you want to learn more about error handling and why the CATCH block looks like it does, see Part One in my series *Error and Transaction Handling in SQL Server*.

There are two local variables. As you might guess, **@sql** is where we will build our query. **@lineend** is akin to **@nl** that I introduced in the section *Spacing and Formatting*, but in addition to the CR-LF, there is also a comma. Since the lines we generate will end in a comma, it makes perfect sense to have the comma in the variable. This is also due to a restriction with the **string_agg** function that I will return to.

Let's now look at step 1. In this step we set up the **#pivotcols** table. (While you can use any name, I'm using this name as a standard name in this text). Logically, this table needs to have three columns:

1. One that defines the column order, typically called **colno**. This is normally an **int** column.
2. One that specifies the values for the CASE filters, typically called **filterval**. The data type for this column should match the source column it is taken from.
3. One that defines the names of the columns in the result set of the generated query, typically called **colname**. The data type of this column is always **sysname**. (Because it is a metadata name.)

The values in all three columns are always unique.

Sometimes you can use the same column for more than one purpose. In this particular example, **filterval** and **colname** would have the same values, so we skip **filterval** and use **colname** in both places. Had the requirements asked for a strict alphabetic list of the output columns, we could also have skipped **colno**.

This is how we define and fill **#pivotcols** for **ProductCountrySales_sp**:

```
CREATE TABLE #pivotcols (colno   int      NOT NULL PRIMARY KEY,
                         colname sysname  NOT NULL UNIQUE)
INSERT #pivotcols (colno, colname)
   SELECT row_number() OVER(ORDER BY Cou.Continent, Cou.Country), Cou.Country
   FROM   Countries Cou
   WHERE  EXISTS (SELECT *
                  FROM   Customers C
                  JOIN   Orders O ON O.CustomerID = C.CustomerID
                  WHERE  C.Country = Cou.Country
                    AND  O.OrderDate BETWEEN @fromdate AND @todate)
```

I suspect that many readers would have written this as a join over all three tables, but in such case you would have to throw in a DISTINCT to weed out duplicates. Using EXISTS is a cleaner way to achieve the goal. We are in fact asking for countries that had sales in the period.

We now jump to step 5, the part where we set up the body of the query after the SELECT list. It is a good idea to write this part before you do steps 2, 3 and 4, because it is here you determine the selection logic of the query, and it here where you define the aliases you will use in the preceding steps. For this example, this step is not particularly complicated:

```
SELECT @sql += '
  FROM   Orders O
  JOIN   [Order Details] OD ON O.OrderID = OD.OrderID
  JOIN   Customers C ON C.CustomerID = O.CustomerID
  JOIN   Products P ON P.ProductID = OD.ProductID
  WHERE  O.OrderDate BETWEEN @fromdate AND @todate
  GROUP  BY P.ProductName
  ORDER  BY P.ProductName'
```

Note that we are adding to the **@sql** variable with the += operator. (Recall the caveat about using this operator if the string you append is composed of several long strings.)

Now that we have the aliases, we can go back to step 2, which in this example is a one-liner:

```
SELECT @sql = 'SELECT P.ProductName ' + @lineend
```

ProductName is the only column to come before the pivot columns.

Step 3 is the centrepiece of a dynamic pivot from the perspective of dynamic SQL. This where we build the SUM/CASE lines from the contents of **#pivotcols**. Let's first look at how to build a single line, before we look at concatenating them together into the **@sql** variable, so that we are not trying to do too many things at the same time. What we are looking at is to build a line like this:

```
SUM(CASE Country WHEN 'Brazil' THEN TotAmount END) AS Brazil,
```

There are two occurrences of *Brazil* here. One is a string literal, and the second is a column alias. We have learnt that we should always use variables where this is syntactically possible, so we may think that we should use a variable in place of the string literal. However, this fails us here, because parameters only work when we have a fixed number of them, and we don't know in advance how many we will need. Thus, we need to inline both the string literal and the column alias (which never can be a variable anyway.) When we do this, we need to take the usual precautions to prevent SQL injection and avoid syntactic accidents when the country name includes special characters. We may also want to add some indentation so that the debug output is more readable. This leads to something like this:

```
SELECT concat(space(7), ' SUM(CASE C.Country WHEN ', quotename(colname, ''''),
              ' THEN OD.Amount END) AS ', quotename(colname), @lineend)
FROM   #pivotcols
```

You may note that I use the **concat** function rather than using the + operator. It does not really matter here, but if you have a separate **filterval** column in **#pivotcols** this may be an integer or a date value. In that case, **concat** is more practical because all its inputs are implicitly converted to strings, so the code does not have to be littered with an explicit type cast. The call **space**(7) add seven spaces of indentation.

Note: The **concat** function was introduced in SQL 2012, so if you are still on SQL 2008 or earlier, you need to use the + operator instead.

In both places where **colname** appears, it is wrapped in **quotename** so that we can handle country names such as Costa Rica and Côte d'Ivoire without syntax errors. Since **colname** is of type **sysname**, we don't have to worry that the input value to **quotename** is too long and will result in NULL being returned. We also add **@lineend** to the string to get the final comma and a line-break for pretty output.

To have any use for these lines, we need to concatenate them into a single string. We have done this a few times already in this article, but nowhere it is as essential as with dynamic pivot, so let's take a little closer look at how to do this. There are two options: the more straightforward function **string_agg**, which is available in SQL 2017 and later, and the less-than-intuitive FOR XML PATH, available since SQL 2005. Both of them have some peculiarities that call for some modification to the SELECT above. Here is the complete code for step 3 with **string_agg**:

```
; WITH sumcaselines AS (
   SELECT colno,
          convert(nvarchar(MAX),
             concat(space(7), ' SUM(CASE C.Country WHEN N', quotename(colname, ''''),
                    ' THEN OD.Amount END) AS ', quotename(colname))) AS sumcase
   FROM   #pivotcols
)
SELECT @sql += string_agg(sumcase, @lineend) WITHIN GROUP (ORDER BY colno)
FROM   sumcaselines
```

The reason I use a CTE here is simply to have the string concatenation separated from the call to **string_agg**, but if you prefer to put the concatenation inside **string_agg**, you can do so. You may observe that I have modified the SELECT above in two regards. The first is that I have wrapped the **concat** operation in a **convert** to **nvarchar(MAX)**. If your pivot has many columns, it could be that the total length of the concatenation exceeds 8000 bytes, which means that the result that comes out of **string_agg** must be **nvarchar(MAX)**. However, the output type from **string_agg** is the same as the type of the input, so if the input is **nvarchar**(n), you cannot get more than 4000 characters back. Thus, in order to get **nvarchar(MAX)** back from **string_agg**, you need to send **nvarchar(MAX)** in. (Thankfully, if you forget the **convert** and the result is more than 8000 bytes, **string_agg** raises an error and does not truncate the string silently.) The other modification is that **@lineend** is absent from the main concatenation, but instead it passed as the second argument to **string_agg**, that is, the delimiter for the concatenation. (This argument must be a constant string literal or a variable. It cannot be expression such as `',' + @nl`. Which is one reason why **@lineend** includes the comma)

Also pay attention to the WITHIN GROUP clause which we have not seen in our previous encounters with **string_agg**. This clause permits us specify the order within the concatenated string, and the order is already defined by the **colno** column in **#pivotcols**, so we use that column.

This is the solution with FOR XML PATH:

```
SELECT @sql +=
    (SELECT concat(space(7), ' SUM(CASE C.Country WHEN ', quotename(colname, ''''),
                   ' THEN OD.Amount END) AS ', quotename(colname), @lineend)
     FROM   #pivotcols
     ORDER  BY colno
     FOR XML PATH(''), TYPE).value('.', 'nvarchar(MAX)')

SELECT @sql = substring(@sql, 1, len(@sql) - len(@lineend))
```

string_agg has a special place for the delimiter between the strings and understands that it should not be added after the last string. FOR XML PATH is primarily designed for a different purpose, so it does not have this capability. For this reason, we have to add a second statement to remove the final **@lineend**, since else that comma would give us syntax errors. For the ordering on country name, we use a regular ORDER BY clause. There is no need here to cast to **nvarchar(MAX)** here, but this is done implicitly in this case.

You may wonder what all those funny things on the last line really mean and why you need them. But it is too convoluted to explain for this context, so I will leave you in the dark. You will need to trust me that this works. And, yes, you need all that gobbledygook. But it always looks the same, so you can just copy and paste it when you need it.

Note: Many people who post examples with FOR XML PATH, wrap the FOR XML query in the **stuff** function to remove the trailing delimiter. However, I think this makes the code difficult to read, which is why I prefer to remove **@lineend** in a separate statement.

The last step to look at is step 4 which is very simple in this first example: since we don't have any extra columns, it's empty. We will return to this step in the next example.

Here is the complete code for **ProductCountrySales_sp** with the two alternatives of step 3 within frames. That is, you would only include one of these frames when you create the procedure.

```
CREATE OR ALTER PROCEDURE ProductCountrySales_sp @fromdate date,
                                                 @todate   date,
                                                 @debug    bit = 0 AS
BEGIN TRY
   DECLARE @lineend char(3) = ',' + char(13) + char(10),
           @sql     nvarchar(MAX)

   CREATE TABLE #pivotcols (colno   int       NOT NULL PRIMARY KEY,
                            colname sysname   NOT NULL UNIQUE)
   INSERT #pivotcols (colno, colname)
      SELECT row_number() OVER(ORDER BY Cou.Continent, Cou.Country), Cou.Country
      FROM   Countries Cou
      WHERE  EXISTS (SELECT *
                     FROM   Customers C
                     JOIN   Orders O ON O.CustomerID = C.CustomerID
                     WHERE  C.Country = Cou.Country
                       AND  O.OrderDate BETWEEN @fromdate AND @todate)

   SELECT @sql = 'SELECT P.ProductName ' + @lineend
```

```
   ; WITH sumcaselines AS (
      SELECT colno,
             convert(nvarchar(MAX),
                concat(space(7), ' SUM(CASE C.Country WHEN N', quotename(colname, ''''),
                       ' THEN OD.Amount END) AS ', quotename(colname))) AS sumcase
      FROM   #pivotcols
   )
   SELECT @sql += string_agg(sumcase, @lineend) WITHIN GROUP (ORDER BY colno)
   FROM   sumcaselines
```

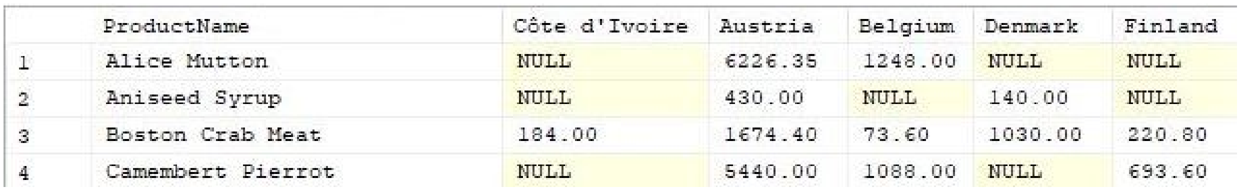
```
  SELECT @sql +=
       (SELECT concat(space(7), ' SUM(CASE C.Country WHEN ', quotename(colname, ''''),
                      ' THEN OD.Amount END) AS ', quotename(colname), @lineend)
        FROM   #pivotcols
        ORDER  BY colno
        FOR XML PATH(''), TYPE).value('.', 'nvarchar(MAX)')

   SELECT @sql = substring(@sql, 1, len(@sql) - len(@lineend))
```

```
   SELECT @sql += '
     FROM   Orders O
     JOIN   [Order Details] OD ON O.OrderID = OD.OrderID
     JOIN   Customers C ON C.CustomerID = O.CustomerID
     JOIN   Products P ON P.ProductID = OD.ProductID
     WHERE  O.OrderDate BETWEEN @fromdate AND @todate
     GROUP  BY P.ProductName
     ORDER  BY P.ProductName'

   IF @debug = 1
      PRINT @sql
   EXEC sp_executesql @sql, N'@fromdate date, @todate date', @fromdate, @todate
END TRY
BEGIN CATCH
   IF @@trancount > 0 ROLLBACK TRANSACTION
   ; THROW
END CATCH
```

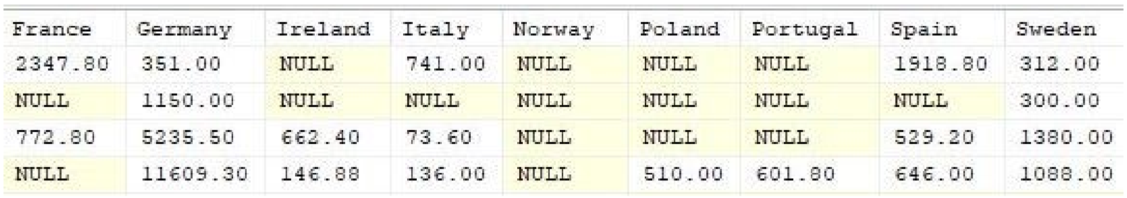
Let's now look at some sample executions. This call covers the entire period that NorthDynamic Traders were active, so this is all orders in the database:

```
EXEC ProductCountrySales_sp '19960101', '19981231', 1
```

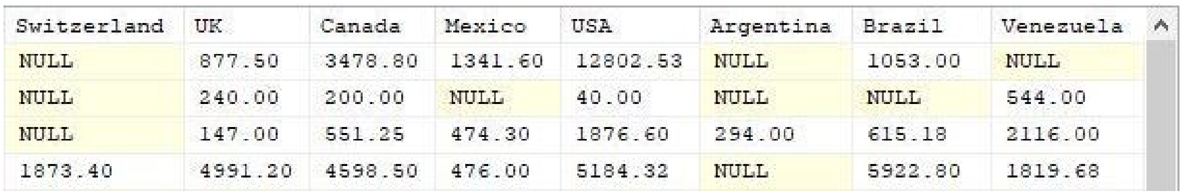
Here is the beginning of the result set, split up over three images to fit the page width:



	ProductName	Côte d'Ivoire	Austria	Belgium	Denmark	Finland
1	Alice Mutton	NULL	6226.35	1248.00	NULL	NULL
2	Aniseed Syrup	NULL	430.00	NULL	140.00	NULL
3	Boston Crab Meat	184.00	1674.40	73.60	1030.00	220.80
4	Camembert Pierrot	NULL	5440.00	1088.00	NULL	693.60



France	Germany	Ireland	Italy	Norway	Poland	Portugal	Spain	Sweden
2347.80	351.00	NULL	741.00	NULL	NULL	NULL	1918.80	312.00
NULL	1150.00	NULL	NULL	NULL	NULL	NULL	NULL	300.00
772.80	5235.50	662.40	73.60	NULL	NULL	NULL	529.20	1380.00
NULL	11609.30	146.88	136.00	NULL	510.00	601.80	646.00	1088.00



Switzerland	UK	Canada	Mexico	USA	Argentina	Brazil	Venezuela
NULL	877.50	3478.80	1341.60	12802.53	NULL	1053.00	NULL
NULL	240.00	200.00	NULL	40.00	NULL	NULL	544.00
NULL	147.00	551.25	474.30	1876.60	294.00	615.18	2116.00
1873.40	4991.20	4598.50	476.00	5184.32	NULL	5922.80	1819.68

You can also look in the *Messages* tab to see the generated query. Here is an abbreviated version:

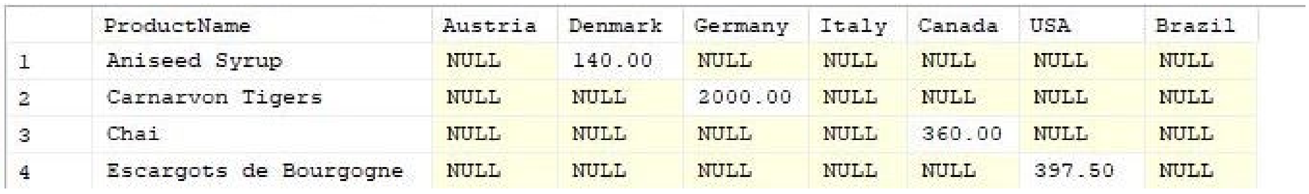
```
SELECT P.ProductName ,
        SUM(CASE C.Country WHEN 'Côte d''Ivoire' THEN OD.Amount END) AS [Côte d'Ivoire],
        SUM(CASE C.Country WHEN 'Austria' THEN OD.Amount END) AS [Austria],
        SUM(CASE C.Country WHEN 'Belgium' THEN OD.Amount END) AS [Belgium],
        ...
        SUM(CASE C.Country WHEN 'Brazil' THEN OD.Amount END) AS [Brazil],
        SUM(CASE C.Country WHEN 'Venezuela' THEN OD.Amount END) AS [Venezuela]
     FROM   Orders O
     JOIN   [Order Details] OD ON O.OrderID = OD.OrderID
     JOIN   Customers C ON C.CustomerID = O.CustomerID
     JOIN   Products P ON P.ProductID = OD.ProductID
     WHERE  O.OrderDate BETWEEN @fromdate AND @todate
     GROUP  BY P.ProductName
     ORDER  BY P.ProductName
```

Pay attention to the first country, Côte d'Ivoire (recall that countries are primarily sorted by continents, so Africa comes first). Without **quotename**, that country name would have produced syntax errors, due to the single quote and the space in the name.

If you run it for only two weeks in July 1997, you get far less columns:

```
EXEC ProductCountrySales_sp '19970701', '19970714', 1
```

This is the first four rows in the result set, which only has six columns:



	ProductName	Austria	Denmark	Germany	Italy	Canada	USA	Brazil
1	Aniseed Syrup	NULL	140.00	NULL	NULL	NULL	NULL	NULL
2	Carnarvon Tigers	NULL	NULL	2000.00	NULL	NULL	NULL	NULL
3	Chai	NULL	NULL	NULL	NULL	360.00	NULL	NULL
4	Escargots de Bourgogne	NULL	NULL	NULL	NULL	NULL	397.50	NULL

And here is the SELECT list:

```
SELECT P.ProductName ,
        SUM(CASE C.Country WHEN 'Austria' THEN OD.Amount END) AS [Austria],
        SUM(CASE C.Country WHEN 'Denmark' THEN OD.Amount END) AS [Denmark],
        SUM(CASE C.Country WHEN 'Germany' THEN OD.Amount END) AS [Germany],
        SUM(CASE C.Country WHEN 'Italy' THEN OD.Amount END) AS [Italy],
        SUM(CASE C.Country WHEN 'Canada' THEN OD.Amount END) AS [Canada],
        SUM(CASE C.Country WHEN 'USA' THEN OD.Amount END) AS [USA],
        SUM(CASE C.Country WHEN 'Brazil' THEN OD.Amount END) AS [Brazil]
     FROM   Orders O
     ...
```

We will now move on to the next example, which is **ProductEmployeeSales_sp**. In this procedure we want to see sales per product and employee during a period, with the employees as columns, sorted by their total sales. We should only have columns for employees who actually had sales in the period. In this example, we also want grand totals, both per product and per employee.

This leads to a couple of changes from the previous example. To start with, we need a separate **filterval** column in **#pivotcols**, since we want to filter by **EmployeeID**, whereas for the column names we want the names of the employees. Next, because of the **Grand Total** column, step 4 is non-empty. As it turns out, this step does not manifest itself by a separate line of code, but rather to modifications to the code for steps 3 and 5. The **Grand Total** column itself is simply added first to the segment with the FROM-JOIN clauses. And because of this column, the last dynamic column should now be followed by a comma. In the solution with **string_agg**, this means that we need to add an extra **@lineend**. On the other hand, in the variant with FOR XML PATH, this means that we should leave out the **substring** operation that removes the final **@lineend**, so this part is actually shorter when we have any extra columns.

Since we also want grand totals per employees, we need to use the GROUPING SETS clause in the GROUP BY clause, in the same manner as in the previous example with a static query. (Which, as noted, has little to do with dynamic pivot as such, but since it is not uncommon to ask for it, I have added this to the example.)

Here is the full code, with the two variations for step 3 in frames. I have also highlighted the differences with regards to the first example.

```
CREATE OR ALTER PROCEDURE ProductEmployeeSales_sp @fromdate date,
                                                  @todate   date,
                                                  @debug    bit = 0 AS
BEGIN TRY
   DECLARE @lineend char(3) = ',' + char(13) + char(10),
           @sql     nvarchar(MAX)

   CREATE TABLE #pivotcols (colno     int      NOT NULL PRIMARY KEY,
                            filterval int      NOT NULL UNIQUE,
                            colname   sysname  NOT NULL UNIQUE)

   ; WITH OrderAgg AS (
       SELECT EmployeeID, SUM(OD.Amount) AS Amount
       FROM   Orders O
       JOIN   [Order Details] OD ON O.OrderID = O.OrderID
       WHERE  OrderDate BETWEEN @fromdate AND @todate
       GROUP  BY O.EmployeeID
   )
   INSERT #pivotcols(colno, filterval, colname)
      SELECT row_number() OVER(ORDER BY OA.Amount DESC, E.EmployeeID), E.EmployeeID,
             E.FirstName + ' ' + E.LastName
      FROM   Employees E
      JOIN   OrderAgg OA ON E.EmployeeID = OA.EmployeeID
      SELECT @sql = 'SELECT P.ProductName ' + @lineend
```

```
      ; WITH sumcaselines AS (
         SELECT colno,
                convert(nvarchar(MAX),
                   concat(space(7), ' SUM(CASE O.EmployeeID WHEN ', filterval,
                       ' THEN OD.Amount END) AS ', quotename(colname))) AS sumcase
         FROM   #pivotcols
      )
      SELECT @sql += string_agg(sumcase, @lineend) WITHIN GROUP (ORDER BY colno) +
                     @lineend
      FROM   sumcaselines
```

```
      SELECT @sql +=
         (SELECT concat(space(7), ' SUM(CASE O.EmployeeID WHEN ', filterval,
                       ' THEN OD.Amount END) AS ',  quotename(colname), @lineend)
         FROM   #pivotcols
         ORDER  BY colno
         FOR XML PATH(''), TYPE).value('.', 'nvarchar(MAX)')
    -- Nothing here.
```

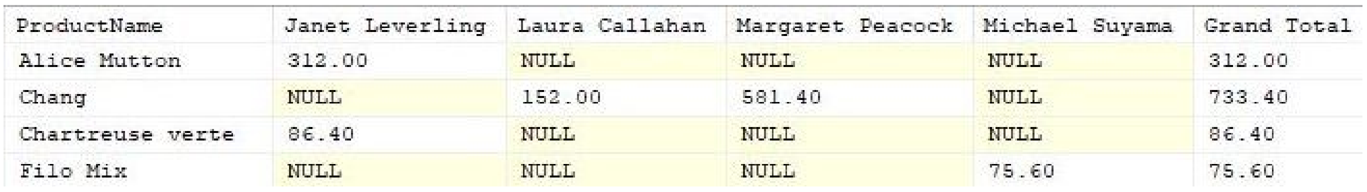
```
      SELECT @sql += '
                  SUM(OD.Amount) AS [Grand Total]
          FROM    Orders O
          JOIN    [Order Details] OD ON O.OrderID = OD.OrderID
          JOIN    Products P ON P.ProductID = OD.ProductID
          WHERE   O.OrderDate BETWEEN @fromdate AND @todate
          GROUP   BY GROUPING SETS (P.ProductName, ())
          ORDER   BY GROUPING (P.ProductName), P.ProductName'

      IF @debug = 1
         PRINT @sql
      EXEC sp_executesql @sql, N'@fromdate date, @todate date', @fromdate, @todate
   END TRY
   BEGIN CATCH
     IF @@trancount > 0 ROLLBACK TRANSACTION
     ; THROW
   END CATCH
```

You may note that there is a CTE when building **#pivotcols**. This CTE computes the sales amount for the employees during the period in a relational way, so we can use this sales amount to order the columns.

Here is an example call:

```
   EXEC ProductEmployeeSales_sp '19970201', '19970214', 1
```

It lists four (out of nine) Employees. Here are the first few rows in the result set

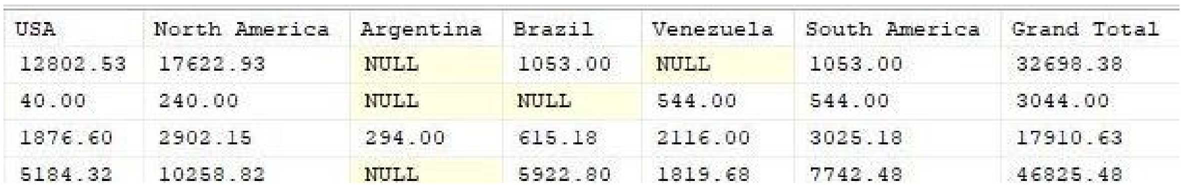


ProductName	Janet Leverling	Laura Callahan	Margaret Peacock	Michael Suyama	Grand Total
Alice Mutton	312.00	NULL	NULL	NULL	312.00
Chang	NULL	152.00	581.40	NULL	733.40
Chartreuse verte	86.40	NULL	NULL	NULL	86.40
Filo Mix	NULL	NULL	NULL	75.60	75.60

The last example is **ProductContinentSales_sp**. It is similar to **ProductCountrySales_sp**, but here we want sub-totals per continent as well as a grand total. To simplify things, we ignore whether a country has sales in the period, but we include all countries from which there is at least one customer. I mainly leave the example uncommented, but I will point out that **filterval** is again absent from **#pivotcols**, but on the other hand there is a help column which is used in step 3 where we want to filter on the country name for the regular columns, whereas for the sub-total columns we want to filter on the continent name. I show this step with FOR XML only.

```
   CREATE OR ALTER PROCEDURE ProductContinentSales_sp @fromdate date,
                                                       @todate   date,
                                                       @debug    bit = 1 AS
   BEGIN TRY
      DECLARE @lineend char(3) = ',' + char(13) + char(10),
              @sql nvarchar(MAX)
   CREATE TABLE #pivotcols (colno       int       NOT NULL PRIMARY KEY,
                            colname     sysname   NOT NULL UNIQUE,
                            IsContinent bit       NOT NULL)

   INSERT #pivotcols (colno, colname, IsContinent)
      SELECT row_number() OVER(ORDER BY Continent, IsContinent, Country),
             IIF(IsContinent = 1, Continent, Country), IsContinent
      FROM   (SELECT DISTINCT C.Country, cou.Continent, 0 AS IsContinent
              FROM   Customers C
              JOIN   Countries cou ON C.Country = cou.Country
              UNION ALL
              SELECT DISTINCT NULL, cou.Continent, 1 AS IsContinent
              FROM   Customers C
              JOIN   Countries cou ON C.Country = cou.Country) AS u

   SELECT @sql = 'SELECT P.ProductName ' + @lineend

   SELECT @sql +=
       (SELECT concat(space(7), ' SUM(CASE ',
                       IIF(IsContinent = 1, 'cou.Continent', 'C.Country'),
                       ' WHEN ', quotename(colname, ''''),
                       ' THEN OD.Amount END) AS ', quotename(colname), @lineend)
        FROM   #pivotcols
        ORDER  BY colno
        FOR XML PATH(''), TYPE).value('.', 'nvarchar(MAX)')

   SELECT @sql += '
              SUM(OD.Amount) AS [Grand Total]
       FROM   Orders O
       JOIN   [Order Details] OD ON O.OrderID = OD.OrderID
       JOIN   Customers C  ON C.CustomerID = O.CustomerID
       JOIN   Countries cou ON cou.Country = C.Country
       JOIN   Products P ON P.ProductID = OD.ProductID
       WHERE  O.OrderDate BETWEEN @fromdate AND @todate
       GROUP  BY GROUPING SETS (P.ProductName, ())
       ORDER  BY GROUPING(P.ProductName), P.ProductName'

   IF @debug = 1
      PRINT @sql
   EXEC sp_executesql @sql, N'@fromdate date, @todate date', @fromdate, @todate
END TRY
BEGIN CATCH
   IF @@trancount > 0 ROLLBACK TRANSACTION
   ; THROW
END CATCH
go
EXEC ProductContinentSales_sp '19960101', '19981231', 1
```

As the result set is fairly wide, I don't include it full here, but I only show the last couple of columns which include two continent totals as well as the grand total.



USA	North America	Argentina	Brazil	Venezuela	South America	Grand Total
12802.53	17622.93	NULL	1053.00	NULL	1053.00	32698.38
40.00	240.00	NULL	NULL	544.00	544.00	3044.00
1876.60	2902.15	294.00	615.18	2116.00	3025.18	17910.63
5184.32	10258.82	NULL	5922.80	1819.68	7742.48	46825.48

7. Situations Where You (Probably) Should Not Use Dynamic SQL

In this chapter I will discuss situations where you may be inclined to use dynamic SQL, but where this may not be the best of choices. It could be that you have made a mistake earlier in your design, or it could also be that there is an alternative which does not require dynamic SQL.

7.1 Dynamic Table Names in Application Code

It's not uncommon to see questions in forums where the poster says that the table name for a query in a stored procedure has to be dynamic. If you feel the urge to do this for something that will run as a regular part of an application, it is very likely that you are barking up the wrong tree. To clarify, I'm talking about queries – SELECT, INSERT, UPDATE, DELETE, MERGE. For something that involves DDL it may be a different thing, but you rarely have DDL running as part of the daily operations of an application.

Most likely, this is the result of some mistake earlier in the design. In a relational database, a table is intended to model a unique entity with a distinct set of attributes (that is, the columns). Typically, different entities require different operations. So if you think that you want to run the same query on multiple tables, something is not the way it should be.

Through the years, I have observed more than one reason why people want to do this, and in this section I will cover a couple of misconceptions. If you have encountered a situation where you wanted to use a dynamic table names in *application code* that I don't cover here, please drop me a line on esquel@sommarskog.se.

Process-Unique Temp Tables

This is a confusion of very inexperienced SQL users, who learnt dynamic SQL far too early. They want a work table that is unique to the current user, so they go on and create a table like **WorkTable_98** where 98 is the spid of the current process and all access to that table has to be through dynamic SQL. Most readers of this text know that the real solution is to use a local temp table, one with a name starting with a single number sign (#). Each such table is only visible for the current process, and it is automatically dropped when the scope where it was created exits. Thus, there is no need to create dynamically named tables in this case.

The Over-Zealous Will to Generalise

It is not uncommon in a database that there are many lookup tables that all look the same: there is an id, a name and maybe some generic auditing columns. This leads some people to think that they should generalise this, so instead of having one set of CRUD (Create, Read, Update, Delete) procedures for each table, they want a single set where the table name is a parameter.

It follows from what I said above that this is wrong. The tables model different entities, and there should be one set of procedures for each table exactly for this reason. If you feel that this means a lot of boring coding, you can write a program that generates these procedures, so adding a new lookup table is a breeze. My experience is that such tables will not always remain carbon copies of each other. That is, during the life of the system additional attributes will be identified for some of these lookup tables, so by time these procedures will deviate from each other simply because they should be different.

A side note here: if you don't use stored procedures at all but create your SQL statements in client code, it is a whole different thing. Here you can have a generic class for all these lookup tables that has the common generic code, and you can instantiate subclasses for each lookup table, adding extra properties for extra columns, overriding methods etc. But T-SQL does not lend itself to those kinds of things.

I don't know, but maybe some people who want do this think they are saving resources by reducing the number of stored procedures. But keep in mind that each batch of dynamic SQL is a nameless stored procedure, so it is all the same. If you have a hundred procedures for a hundred tables, or if you have a single procedure for your hundred tables, that procedure produces a hundred nameless stored procedures producing a hundred query plans, just like a hundred static procedures do.

Multi-Tenant Applications

When you design a multi-tenant application there are a couple of possibilities. One is to have a single database, and have a **TenantID** in every table. Then you carry the tenant ID around in all your stored procedures. This is by no means an uncommon design. It does come with a risk, though: if a casual developer messes up, the data for one tenant can be exposed for another tenant.

For this reason, some people prefer to have one database (or one schema) per tenant. But then they think that they only want one set of stored procedures which takes the name of a tenant database as a parameter, "because everything else would be a maintenance nightmare". I submit that this is totally wrong. Having this single set of stored procedures is a total maintenance nightmare, since about every line of code has to be littered with dynamic SQL. No, you should have a set of stored procedures in every database (or every schema), and then you need to develop methods to distribute and roll out code and table changes to all databases or schemas. In the days of DevOps this should not be overly tricky.

Note: Obviously, if you don't use stored procedures at all, but submit all queries from client code, this scenario is quite simple. For a multi-database application, the database is determined by the connection string. For a multi-schema application, you would probably have personal logins where each user has the appropriate default schema.

Attribute as Part of the Table Name

In this pattern it is very clear that it is a case of multiple tables modelling the same entity. Instead of a single **Orders** table, there is **Orders_ALFKI**, **Orders_VINET** etc with one table per customer. Or **Orders_1996**, **Orders_1997** etc with one table per year. That is, what really is an attribute of the entity and should be a column in a single table has instead been added as part of the table name. This is rarely, if ever, a correct design.

The exact train of thoughts that leads to this design is not fully clear to me. But maybe the people who use this pattern have not fully absorbed the relational way of thinking. This forum thread can serve as an illustration. In this particular case, it appears that it never occurred to the poster that his **CodNeg** should be a column in his temp table.

Home-Brewed Partitioning

It appears that sometimes when people put a key value in the table name, they have a concern that single table would be too big for SQL Server. That anxiety is not very well-founded when it comes to regular day-to-day operations. With suitable indexes you can quickly find the data you are looking for, and it does not matter if the total table size is 1 TB. And if you need to scan all that data, you need to scan it, no matter if it is one table or hundred tables. But it is easier to write the query if it is a single table or a view.

That said, there are situations where you may want to partition your data. Here are some possible reasons:

- You want to distribute the data on multiple filegroups to spread the load, possibly putting older data on slower drives.
- You want to be able age old data quickly. For instance, say that you want to retain all orders for 12 months. So when June 2020 begins, you want to get rid of all orders from June 2019, and you don't want to run a DELETE statement, which would take a long time to complete.
- A fairly advanced scenario: different customers have very different patterns, so you want different plans for the same query depending on the customer, and to that end you want different histograms in the statistics for different customers.

But in none of these cases it makes sense to create **Orders_201906**, **Orders_201907** etc, or **Orders_ALFKI**, **Orders_VINET** etc and then use dynamic SQL to access the table you need for the moment. To wit, SQL Server provides not one, but two different partitioning features: partitioned tables and partitioned views. I will give a very quick overview of them here and then provide links for further reading.

Partitioned tables is the more popular feature. They were introduced in SQL 2005 and for a long time they were only available in Enterprise Edition, but this changed with the release of SQL 2016 SP1, and they are now available in all editions. To create a partitioned table, you first create a partition function to define the ranges for the partitions. Then you define a partition scheme that maps the partition ranges to different filegroups, and finally you create the table on that partition scheme, using it as if it was a filegroup. The feature supports up to 15 000 partitions for a single table.

Partitioned views have been around even longer; they were introduced already in SQL 2000, and they have always been available in all editions of SQL Server. A partitioned view consists of a number of individual tables that all have a common structure and all have the same primary-key columns. For the partitioning column, which should be part of the PK, each table has a CHECK constraint defined so that a certain value fits into at most one table. The view itself consists of a number of SELECT queries against these tables united with UNION ALL. With partitioned views, you would indeed have tables like **Orders_201906** or **Orders_ALFKI**, but all access would be through the view **Orders**. When you query the view, the plan includes all tables in the view, but the plan is such that only the tables qualified by the partitioning key are actually accessed at run-time. I have not seen any upper limit for the number of tables you can have in a partitioned view, but I would say that 100 would be *very* many, and in practice you should stay below 20.

A very specialised form of partitioned views are distributed partitioned views, where the tables are on multiple SQL Server instances. This is quite an advanced feature, and it is only supported in Enterprise Edition.

Normally, all tables in a partitioned view have the same set of columns, but there is room for some flexibility, so if you find that you need new columns in your **Orders** view starting from January 2021, you can leave the tables **Orders_202012** and earlier unchanged and in just put NULL in the SELECT from these tables in the view definition. Or if you need to have extra columns for the customer ALFKI, and only that customer, you can have those columns in **Orders_ALFKI**, and not include them in the view. Queries that access these columns would access **Orders_ALFKI** directly and not the view **Orders**.

Of the three bullet points I listed above, partitioned tables and partitioned views both support the first two, but you can only achieve the last bullet point with partitioned views where each table has its own histogram. (Although it may be possible to get good-enough results with filtered statistics when the data is in a single table.)

When it comes to the second point, partition switching, this is an operation that requires multiple steps, not the least if more than one (logical) table is involved, for instance both an **Orders** and an **OrderDetails** table. If this is going to happen on the 1st of each month, you certainly want to automate that operation in a stored procedure that you schedule. No matter whether you use partitioned tables or partitioned views, this procedure will need to use a whole lot of dynamic SQL to create new a partition or a new table and to get rid of the old one. I'm not including an example of how to do this here, but I like to point out that this is an example of *good* use of dynamic SQL. You have dynamic SQL to perform a specific task inside a single stored procedure. Compare that with dynamic table names where you litter the code all over the place.

This was a very brief introduction to partitioning in SQL Server. For a somewhat longer introduction with code samples etc, Data Platform MVP Catherine Wilhemsen has a two-part blog post *Table Partitioning in SQL Server* which covers partitioned tables. Stefan Delmarco has a very good article on partitioned views, *SQL Server 2000 Partitioned Views*. (This article is a little old, but the feature has not changed much since.)

7.2 Dynamic Column Names

And then there are users who mysteriously think that SQL Server can read their minds, so if they do

```
SELECT @colname FROM tbl
```

SQL Server will actually return the column of which the name is in **@colname**. Common is also that they want to do:

```
UPDATE tbl
SET    @colname = @value
WHERE  keycol = @id
```

But this will only set the variable **@colname** to the value of **@value** – if there is a row in the table where **keycol** is equal to **@id**.

If you recall what I said in the previous section, you know why this is wrong: each column is supposed to model a unique attribute of the entity that the table models. For this reason, different attributes (i.e., columns) tend to have different domains for the values they can take, which makes the above quite a bit pointless.

Thus, I would submit that if you feel that you need to something like the above, there is an error somewhere in the design. One situation I can think of is that you have a table with columns like **SalesAmt_Jan**, **SalesAmt_Feb** etc. The proper design would of course be to have a sub-table **SalesAmts** with one row per month. However, you may be in the unfortunate situation that you are stuck with a design like this. In that case, my preference is to use a IIF expression, how boring it may be:

```
UPDATE tbl
SET    SalesAmt_Jan = IIF(@colname = 'SalesAmt_Jan', @salesamt, SalesAmt_Jan),
       SalesAmt_Feb = IIF(@colname = 'SalesAmt_Feb', @salesamt, SalesAmt_Feb),
       ...
FROM   tbl
WHERE  keycol = @id
```

On SQL 2008 and earlier, you would have to use CASE instead of IIF.

Another misconception I have occasionally sensed is that some developers are under the impression that updating twenty columns will use more resources than updating a single one. A normal UPDATE statement in a stored procedure that supports a UI may look something like this:

```
UPDATE tbl
SET    col1 = @val1,
       col2 = @val2,
       ...
       col20 = @val20
WHERE  keycol = @id
```

Now, say that the user actually only changed the value for **col1** and the variables from **@val2** to **@val20** hold the current values for **col2** to **col20**. In this case they think this is leaner on the server:

```
UPDATE tbl
SET    col1 = @val1
WHERE  keycol = @id
```

I actually ran some tests on this, and my conclusion is that if there are no indexes on the columns **col2** to **col20** there is no difference at all. I found that the amount of transaction log produced was exactly the same for the two statements. When any of the extra columns were present in an index, I did see some increase in the amount of transaction log produced. However, since this typically is a matter of single-row updates, the actual impact is likely to be entirely negligible. On the other hand, if you are generating UPDATE statements that only include the columns for which there are changes, you will generate one UPDATE statement for each column combination, and each of these will result in a cache entry, so would be taking up *more* resources this way.

7.3 Set Column Aliases Dynamically

The desire here is do to something like this:

```
SELECT col AS @myname FROM tbl
```

I would guess that the most common scenario is a semi-dynamic pivot. That is, the number of columns is fixed, but you want the header to reflect the contents of the column. As one example, say that you want to return sales per employee for the last three months, with the employees as rows and the months as columns, and you want year-month in the column header. In many cases this is something you can handle in the presentation layer. If you still were to do this in SQL, you can implement this is a dynamic pivot according to the pattern I introduced earlier. But it is in fact possible to achieve this particular goal without dynamic SQL. Insert the result in a temp table with fixed column names, and then rename the columns as desired. Here is a procedure to do this for the given example. While the conditions are a little more complex, you can recognise the pattern of SUM with CASE filters from the section on dynamic pivot.

```
CREATE OR ALTER PROCEDURE AmountPerEmployee @yearmonth char(6) AS
BEGIN TRY
   CREATE TABLE #temp (EmployeeID int            NOT NULL PRIMARY KEY,
                       month1     decimal(10, 2) NOT NULL,
                       month2     decimal(10, 2) NOT NULL,
                       month3     decimal(10, 2) NOT NULL)

   DECLARE @month1 char(6) =
              convert(char(6), dateadd(MONTH, -2, @yearmonth + '01'), 112),
           @month2 char(6) =
              convert(char(6), dateadd(MONTH, -1, @yearmonth + '01'), 112),
           @nextmon date = dateadd(MONTH, 1, @yearmonth + '01')

   INSERT #temp (EmployeeID, month1, month2, month3)
```

```
      SELECT EmployeeID,
             SUM(CASE WHEN O.OrderDate >= @month1 + '01' AND
                           O.OrderDate < @month2 + '01' THEN OD.Amount END),
             SUM(CASE WHEN O.OrderDate >= @month2 + '01' AND
                           O.OrderDate < @yearmonth + '01' THEN OD.Amount END),
             SUM(CASE WHEN O.OrderDate >= @yearmonth + '01' AND
                           O.OrderDate < @nextmon THEN OD.Amount END)
      FROM   Orders O
      JOIN   (SELECT OrderID, SUM(Amount) AS Amount
              FROM   [Order Details]
              GROUP  BY OrderID) AS OD ON O.OrderID = OD.OrderID
      WHERE  O.OrderDate >= @month1 + '01'
        AND  O.OrderDate < @nextmon
      GROUP  BY EmployeeID

   EXEC tempdb..sp_rename '#temp.month1', @month1, 'COLUMN'
   EXEC tempdb..sp_rename '#temp.month2', @month2, 'COLUMN'
   EXEC tempdb..sp_rename '#temp.month3', @yearmonth, 'COLUMN'

   SELECT E.FirstName, E.LastName, t.*
   FROM   Employees E
   JOIN   #temp t ON E.EmployeeID = t.EmployeeID
END TRY
BEGIN CATCH
   IF @@trancount > 0 ROLLBACK TRANSACTION
   ; THROW
END CATCH
go
EXEC AmountPerEmployee '199802'
```

Note that this technique requires that you return the columns from the temp table with `SELECT *`; since you don't know the names, you cannot list them.

For only three months, a full dynamic pivot would be a bit of an overkill. On the other hand, if the requirement would be for 24 months, the solution above is not realistic; the thought of 24 month variables is just repelling. You could still stay with the main idea, and have the months in a temp table, and then you loop over that temp table for the renaming part. However, you would still need 24 SUM/CASE rows in the SELECT list, and it is not unlikely that you would prefer a fully dynamic pivot in this case.

7.4 Dynamic Procedure Names

In this case, the programmer is looking into calling a stored procedure of which the name is defined at run-time. Maybe because the database name is dynamic, maybe of some other reason. There is no need for dynamic SQL here, and you have seen the solution several times in this document: EXEC accepts a variable for the procedure name:

```
EXEC @spname
```

Still... I can't escape to note that with some frequency I see even experienced programmers building a dynamic SQL statement to handle this situation.

7.5 Dynamic ORDER BY Condition

If your stored procedure returns a static query, but you want to permit the user to select the order of the rows, the best is to sort data client-side. Not the least because, then you can permit the user to re-sort the data without having to re-run the query. There are several grid controls out there on the market to help you with this.

But if you want to do it in SQL, you can use CASE:

```
ORDER BY CASE @ordercol = thatcol THEN thatcol END,
         CASE @ordercol = thiscol THEN thiscol END,
         ...
OPTION (RECOMPILE)
```

The addition of OPTION (RECOMPILE) can help you to get a query plan which fits best with the selected sort condition.

Admittedly, if you have multiple sort columns and you also want to permit the user to select between ASC and DESC, this can get out of hand, and a solution with dynamic SQL can be less complex to implement. But in such case you must be careful and not just do something like

```
@sql += ' ORDER BY ' + @sortcols
```

since would make you wide open to SQL injection. I briefly discusssed possible approaches earlier in the section *Other Injection Holes*. I my article *Dynamic Search Conditions* I have some more detailed examples.

7.6 SELECT * FROM tbl WHERE col IN (@list)

This is one of the most common misconceptions among inexperienced SQL users. They do something like:

```
SELECT @list = '1,2,3'
SELECT * FROM tbl WHERE col IN (@list)
```

And then they wonder why it does not work, when they get no rows back. The answer is that it does work. If there is a row where the value of **col** is *1,2,3*, they will get that row back. But rows where **col** = 1, no. To wit, `col IN (1, 2, 3)`, is just a syntactical shortcut for

```
col = 1 OR col = 2 OR col = 3
```

and `col IN (@list)` is the same as `col = @list`.

Thankfully it is not as common as it used to be, but you can still see people who answer such questions with suggesting dynamic SQL. Which is a tremendously poor solution for this problem. Opens for SQL injection, makes the code more complex, and nor is it good for query performance.

The correct solution is to use a list-to-table function, and there are plenty of them out there. Starting with SQL 2016, there is even a built-in function, **string_split**, for the task, although it is not always the best choice. See further my article *Arrays and Lists in SQL Server*.

7.7 Sending in a WHERE Clause as a Parameter

This is something I largely covered already in the section *Other Injection Holes* in the chapter on SQL injection, but I'd like to make a short recap as an example of dynamic SQL abuse.

I have occasionally seen users who want to send in a piece of SQL syntax to a stored procedure, and I don't mean a small piece like a column list for an ORDER BY clause, but a more substantial part like a full WHERE clause. Let me make this perfectly clear: as long as we are talking about plain application code, this is an absolute no-no.

There may be situations where the requirements are more dynamic than you can fit into the fairly inflexible T-SQL syntax, and you rather do the work client-side. And that's alright. If you want to permit users to select filters, what columns to return, what to aggregate on etc, I think this is best solved client-side. But then you should do *all* work in the client for this particular function, and not pass SQL syntax to a stored procedure. Not only would this stored procedure be open to SQL injection, but you would also create a very tight coupling between client and database which will cause a maintenance nightmare.

7.8 Working with a Table with Unknown Columns

Every once in a while, I see these posters on the forums who are unhappy because the temp table they created inside their dynamic SQL is no longer there when the dynamic SQL exits. You tell them that they need to create the temp table before running the dynamic SQL batch, thinking that they use SELECT INTO solely because they are lazy bums. But, no, they reply they can't do that, because their temp table is dynamic.

I have quite a problem with addressing their underlying fallacy, because despite that I've seen this pattern many times over the years, I have never been able to fully understand how they ended up in this miserable situation. When I have asked these people to describe their problem from start to end, I have at best gotten a fragmentary reply, and the thread has died unresolved.

I have a theory, though, and that is that their dynamic temp table is the result of a dynamic pivot. As I said in the section on dynamic pivot, this is non-relational operation, and once you have performed your dynamic pivot, there is only one thing you can do with the result: return the data to the client. You simply are not in Kansas anymore, Toto. If you want to perform some filtering, computations or whatever on your data after the dynamic pivot, you need to go back to the drawing board and rearrange your computation order and perform these actions before you do the dynamic pivot. The dynamic pivot must always be the last thing in the SQL part of your solution. Remaining work will have to take place client-side.

One scenario that I have seen some poster allude to was that he was loading files, and the files had different layout every time. That alone sounds suspicious, but if you really have such a situation, it does not like a good task for an all-SQL solution, but this should be orchestrated from a client-side language or something like SSIS (SQL Server Integration Services).

All that said, if you absolutely want a temp table with dynamic columns, there is a way to do it. Create a stub table with your fixed columns. Then you add the dynamic columns with ALTER TABLE ADD inside the dynamic SQL. I'm not showing an example, because it is definitely nothing that I recommend. And it's certainly nothing for users with low experience of SQL, since you will need to know how derive the metadata from the input of the temp table, and this is a trick for expert users. I'm tempted to say that you would only enter this path, if you really need to love to hurt yourself.

7.9 BACKUP/RESTORE

Since I have said dynamic SQL is a fair game for DBA tasks, you may be surprised to see BACKUP/RESTORE in this chapter, but there is a simple reason. It is not uncommon to see code like this:

```
DECLARE @db         sysname = 'Northwind',
        @backuppath nvarchar(200) = 'C:\temp',
        @sql        nvarchar(MAX)

SELECT @sql = 'BACKUP DATABASE ' + quotename(@db) +
              ' TO DISK = ''' + @backuppath + '\' + @db + '-' +
              replace(convert(varchar(19), sysdatetime(), 126), ':', '') + '.bak'''
PRINT @sql
EXEC(@sql)
```

So what's wrong here? Because you can do it this simple:

```
DECLARE @db         sysname = 'NorthDynamic',
        @backuppath nvarchar(200) = 'C:\temp',
        @filepath   nvarchar(250)

SELECT @filepath = @backuppath + '\' + @db + '-' +
                replace(convert(varchar(19), sysdatetime(), 126), ':', '') + '.bak'
BACKUP DATABASE @db TO DISK = @filepath
```

That is, BACKUP and RESTORE accept variable for most or all of their arguments. Here is a working example for RESTORE:

```
DECLARE @db       sysname = 'BigDB',
        @dmpname  nvarchar(250) = 'S:\MSSQL\BigDB.bak',
        @dev1     sysname = 'BigDB',
        @mdfname  nvarchar(250) = 'S:\MSSQL\SJUTTON\BigDB.mdf',
        @dev2     sysname = 'BigDB_log',
        @ldfname  nvarchar(250) = 'S:\MSSQL\SJUTTON\BigDB.ldf',
        @stats    int = 10

RESTORE DATABASE @db FROM DISK = @dmpname
WITH MOVE @dev1 TO @mdfname,
     MOVE @dev2 TO @ldfname,
     STATS = @stats
```

(Change and database and file names to something you have on your server to test.)

I have not verified each and every piece of the syntax for the BACKUP and RESTORE commands. The syntax graphs in Books Online indicate that variables are supported in most places, but there are also parts where variables do not appear in the syntax. For instance, Books Online does not say that variables are supported with MOVE or STATS, but as you can see in the example above, they are accepted.

That said, there are situations where you will need to use dynamic SQL for these commands. If you want to make it a parameter whether to use COMPRESSION, INIT, FORMAT or any other such keyword, you will need to build a dynamic string – but you can still pass database and the name of the backup file as parameters to **sp_executesql**.

Another situation where you need to use dynamic SQL is if you want to use a striped backup. As long as the number of stripes are fixed, there is no problem to do it statically:

```
BACKUP DATABASE @db TO DISK = @stripe1, @stripe2, @stripe3
```

But this breaks down, if want a variable number of stripes depending on the database size or something else, since the shape of the string depends on the how many stripes you want. You *can* still use **sp_executesql** to execute the command, if you settle on a max number of stripes, say 50, and always pass **@stripe1**, **@stripe2** up to **@stripe50**, even if not all these **@stripe** variables appear in the BACKUP command you generate. No, I don't really expect you to do this, and nor would I do so myself, but I would build a BACKUP command with all the stripes inlined, and presumably I would have the name of the stripes (or some part of the name) in a table somehow. And I would of course take precautions to avoid SQL injection and wrap all stripes in **quotename** or **quotestring**.

A similar case exists with RESTORE WITH MOVE. If you want to handle databases with any number of files, you will need to work with dynamic SQL since you don't know beforehand how many MOVE clauses you will need.

But for the plain-vanilla everyday use of these commands, there is no need to use dynamic SQL. Simply use variables.

- -

8. Conclusion

You have now read an article about dynamic SQL, and my hope is that you have learnt there are both good and bad things you can do with dynamic SQL. Dynamic SQL can be a blessing to automate DBA tasks, and dynamic SQL used in the right place can be a blessing in application code too. But applied incorrectly, dynamic SQL can be a curse, producing code that is almost impossible to read and maintain and which causes performance issues and opens security vulnerabilities. And it is not uncommon that an urge to use dynamic SQL is due to a mistake earlier in the design where you have violated one or more of the basic rules for relational databases.

The very most important points in this article in summary:

- Always use parameterised statements. Never inline a value that can be passed as a parameter!
- When you build strings with object names in variables: always use **quotename**.
- When you need to inline string values, for instance in DDL statements, use **quotename** with single quote as the second parameter, if you know that the value is <= 128 characters. Else use **dbo.quotestring** or **dbo.quotestring_n**.
- Beware of SQL injection!
- Always include this line when you work with dynamic SQL:

  ```
  IF @debug = 1 PRINT @sql
  ```

  Always!

8.1 Acknowledgement and Feedback

My articles do not exist in a vacuum, and I receive many valuable comments from MVP colleagues, readers of my articles and other people. For this article I would like to thank the following persons for their input: Anthony Faull, Chintak Chhapia and Jonathan Van Houtte. (Yes, this is a short list, because the article is new in its current shape. I am sure that it will be extended by time.)

If you have comments on the article, you are more than welcome to mail me on esquel@sommarskog.se. Your comment may be on the actual contents, but you are also more than welcome to point out spelling and grammar errors. Or just point out passages where I could explain things better. On the other hand, if you have a specific problem that you are struggling with, I encourage you to ask your question in a public forum, since it is very likely that you will get help faster that way, as more people will see your post. If you want me to see the question, the best place is the Transact-SQL forum on MSDN, but I should warn you that time usually does not permit me to read all posts there.

9. Revisions

2020-01-12
Jonathan Van Houtte was kind to suggest a tip for the section *The Necessity of Debug Prints* for one more way to display long SQL strings.

2020-01-11
First version. (Yes, there was an article with the same title at the same URL, but this is essentially an entirely new article on the same subject.)

Back to my home page.