

Title: Dynamic SQL
Cat: Programming Standards
Date: *07 December 2001*
Author: *Christophe Perdreau*

1 Overview.....	4
1.1 Subject.....	4
1.2 Summary.....	4
1.3 Audience.....	4
1.4 Limitations.....	4
2 Introduction to Dynamic SQL.....	5
2.1 Static/Pre-compiled SQL.....	5
2.2 Dynamic-SQL.....	6
2.3 When to use Dynamic –SQL.....	7
3 Dynamic SQL.....	7
3.1 Building the query.....	7
3.2 DATETIME and SMALLDATETIME entities.....	8
3.3 Simulating a SELECT @MyVar = value FROM ...	9
3.4 @@ERROR and @@ROWCOUNT.....	10
3.5 Sp_executesql versus exec() or execute().....	10
3.6 Stored Procedures.....	10
4 Pitfalls.....	11
4.1 Runtime parsing.....	11
4.2 @MyQuery.....	11
4.3 CONVERT function.....	11
5 Other methods.....	11
5.1 Commonly used method.....	12
5.2 Microsoft’s recommended method.....	12

1 Overview

1.1 Subject

This document establishes the programming standards for Dynamic SQL.

1.2 Summary

After a brief introduction to the subject we will review the programming standards. We will then discuss various pitfalls to be aware of. And finally we will emphasize the downsides with 2 other documented ways of implementing Dynamic SQL.

1.3 Audience

All SQL/DB developers should read this document.

1.4 Limitations

These standards apply to Dynamic SQL used within Transact-SQL.

2 Introduction to Dynamic SQL

2.1 Static/Pre-compiled SQL

Most DDL and DML statements inside Transact-SQL code – including Stored Procedures - are static. The term static means:

- a) the complete structure of the statement – i.e. name of table(s), SELECT clause, WHERE clause... - is known at design time
- b) the statement must be successfully parsed and compiled by the SQL engine BEFORE the statement is allowed to run. In particular, all references to tables, columns and other DB objects must be resolved before runtime.
- c) the structure of the statement can not be changed at runtime

E.g.

```
DECLARE
    @MyCustId INT

    SELECT FirstName, Surname
    FROM Customer
    WHERE CustId = @MyCustId
```

In the above statement the columns in the WHERE clause are known at design time. So is the name of the table and the structure of the WHERE clause. Notice the use of the variable @MyCustId. This variable allows basic control of the statement at runtime but it cannot provide a mechanism to alter the structure of the query. For instance it would be nice if the name of the table were also a variable so we could target a specific table at runtime. Let's see what the code to achieve this would look like:

```
DECLARE
    @MyCustId INT,
    @MyTableName VARCHAR(100)

    SET @MyTableName = 'Customer'

    SELECT FirstName, Surname
    FROM @MyTableName
    WHERE CustId = @MyCustId
```

At first the above piece of code looks OK but unfortunately it does not compile. We have the same problem if we try to make the columns in the SELECT clause dynamic. Here what the code would look like:

```
DECLARE
    @MyCustId INT,
```

```
@MyColNames VARCHAR(200)

SET @MyColNames = 'FirstName, Surname'

SELECT @MyColNames
FROM Customer
WHERE CustId = @MyCustId
```

Again the above piece of code does not compile. The only way to implement these two features is through Dynamic-SQL.

2.2 Dynamic-SQL

These limitations imposed by static SQL are fine for most needs and **static SQL should always be considered first**. But there are cases when the actual statement is not fully known until runtime. Or there are certain DDLs – such as CREATE VIEW – that are banned from SP code. In these situations we need Dynamic-SQL.

Dynamic SQL relies on a system Stored Procedure called **sp_executesql**. This SP accepts as its first parameter an NVARCHAR value that represents the statement to execute. For instance:

```
EXEC sp_executesql N'SELECT * FROM CUSTOMER'
```

This can be rewritten as:

```
DECLARE
    @MyQuery NVARCHAR(2000)
BEGIN
    SET @MyQuery = N'SELECT * FROM CUSTOMER'
    EXEC sp_executesql @MyQuery
END
```

This is where the strength of Dynamic-SQL is clearly visible: the content of @MyQuery can be fully adjusted at runtime. We can for instance create an SP that accept a parameter called @TableName that represents the name of the table to SELECT from: The body of the SP could be:

```
DECLARE
    @MyQuery NVARCHAR(2000)
BEGIN
    SET @MyQuery = N'SELECT * FROM ' + @TableName
    EXEC sp_executesql @MyQuery
END
```

This is definitely something you cannot do using pre-compiled SQL statements.

2.3 When to use Dynamic –SQL

Dynamic-SQL is very powerful but should be used ONLY when pre compiled SQL can not provide the functionalities required. Here are a few instances when Dynamic-SQL should be used:

- 1) The target table is unknown at compile time in a SELECT, UPDATE or DELETE – aka DDL
- 2) The columns part of the SELECT clause are unknown at compile time
- 3) The structure of a WHERE clause is unknown during compilation
- 4) The number of SET clauses is unknown during compilation
- 5) You need the ability to issue CREATE VIEW statements within an SP

3 Dynamic SQL

3.1 Building the query

All Transact-SQL making use of Dynamic-SQL should:

- a) declare an NVARCHAR variable called @MyQuery.
- b) the size of this variable should be big enough to accommodate the biggest statement to be built
- c) the statement should initially be built as if it were a normal static SQL statement using @XXX variables wherever required – even for things like table and column names. These @XXX variables must be real variable declared in your code
- d) package the above statement into an NVARCHAR constant using multiple lines if necessary to provide easy reading
- e) assign value to @MyQuery
- f) add a REPLACE statement for each @XXX variable to substitute

Example: we need to write a SELECT * statement against a given table – it's a variable – and for a given CustId.

Steps a) and b)

```
DECLARE
    @MyQuery          NVARCHAR(2000)
```

Step c)

```
SELECT *
FROM @MyTableName
WHERE CustId = @MyCustId
```

Step d)

```
N'SELECT * ' +
N'FROM @MyTableName ' +
```

```
N'WHERE CustId = @MyCustId'
```

Step e)

```
SET @MyQuery = N'SELECT * ' +  
                N'FROM @MyTableName ' +  
                N'WHERE CustId = @MyCustId'
```

Step f)

```
REPLACE (@MyQuery, '@MyTableName', @MyTableName)  
REPLACE (@MyQuery, '@MyCustId' , @MyCustId)
```

And the whole Transact-SQL looks like:

```
DECLARE  
    @MyQuery      NVARCHAR(2000)  
    @MyTableName  VARCHAR(100)  
BEGIN  
    SET @MyTableName = 'Customer'  
  
    SET @MyQuery = N'SELECT * ' +  
                  N'FROM @MyTableName ' +  
                  N'WHERE CustId = @MyCustId'  
  
    SET @MyQuery = REPLACE (@MyQuery, '@MyTableName', @MyTableName)  
    SET @MyQuery = REPLACE (@MyQuery, '@MyCustId' , @MyCustId)  
  
    EXEC sp_executesql @MyQuery  
  
END
```

The REPLACE above works fine with VARCHAR and INT and related data types. For these data types the implicit conversion to NVARCHAR is correct. For other data types explicit conversion is usually required. For instance DATETIME requires an explicit CONVERT.

Also for VARCHAR and related data types make sure you add single quotes at the right place:

```
SET @MyQuery = N'SELECT * ' +  
                N'FROM @MyTableName ' +  
                N'WHERE CustName = ''@MyCustName'''
```

3.2 DATETIME and SMALLDATETIME entities

DATETIME and SMALLDATETIME should be treated with great care to avoid the usual American versus English date format issues. To be safe please use the following REPLACE/CONVERT statement:

```
SET @MyQuery = REPLACE(@MyQuery, '@MyDT',  
                        CONVERT(NVARCHAR, @MyDT, 113))
```

Conversion style 113 is defined as 'dd mon yyyy hh:mm:ss:mmm(24h)'.

Example:

```
SET @MyQuery = N'SELECT * ' +  
              N'FROM @MyTableName ' +  
              N'WHERE CreationDate > ''@MyDT'''  
  
SET @MyQuery = REPLACE(@MyQuery, '@MyDT',  
                        CONVERT(NVARCHAR, @MyDT, 113))
```

3.3 Simulating a SELECT @MyVar = value FROM ...

Sp_executesql executes within its own context and therefore is not aware of any variables showing in your DECLARE block(s). So in Dynamic-SQL it is not possible to implement simple **SELECT @MyVar = value FROM ...** type queries.

Example:

```
DECLARE  
    @MyQuery NVARCHAR(2000),  
    @MyMaxCustId INT  
BEGIN  
    SET @MyQuery = N'SELECT @MyMaxCustId = MAX(CustId) ' +  
                  N'FROM CUSTOMER'  
    EXEC sp_executesql @MyQuery  
END
```

In the above we would expect @MyMaxCustId to acquire the value of MAX(CustId) after executing sp_executesql. But instead you will get an error message saying that @MyMaxCustId is unknown.

The way to remedy this is to use a third party table and column. For instance let's have a table called TabStorage with a SEEDed PK called StorageId and another column called MaxCustId defined as INT:

TabStorage		
StorageId	INT, SEED	

MaxCustId	INT, NULL ALLOWED	

The idea is to add row in this table BEFORE running the SELECT and use the associated MaxCustId column as temporary storage.

```
DECLARE
    @MyQuery NVARCHAR(2000),
    @MyMaxCustId INT,
    @MyStorageId INT
BEGIN
    INSERT INTO TabStorage(MaxCustId) VALUES (NULL)
    SET @MyStorageId = @@IDENTITY

    SET @MyQuery = N'UPDATE TabStorage ` +
        N'SET MaxCustId = (SELECT MAX(CustId) ` +
            N'FROM CUSTOMER) ` +
        N'WHERE StorageId = @MyStorageId'

    SET @MyQuery = REPLACE(@MyQuery, '@MyStorageId', @MyStorageId)

    EXEC sp_executesql @MyQuery

    SELECT @MyMaxCustId = MaxCustId
    FROM TabStorage
    WHERE StorageId = @MyStorageId
END
```

3.4 @@ERROR and @@ROWCOUNT

Both @@ERROR and @@ROWCOUNT work as expected.

```
...
EXEC sp_executesql @MyQuery
SELECT @MyError = @@ERROR,
       @MyCount = @@ROWCOUNT
...
```

3.5 Sp_executesql versus exec() or execute()

The sp_executesql SP and the execute() system function provides similar functionalities although sp_executesql is more powerful – see SQL Server documentation. For our purposes we could use either but for the sake of standardization we should all use sp_executesql.

3.6 Stored Procedures

Stored Procedures represent a special case. If the name of the SP is unknown at compile time **BUT the signature is fixed** then a quick and easy way to execute a particular SP at runtime is:

```
DECLARE
    @MySpName VARCHAR(50),
    @MyCustId INT,
    @MyInstId INT

    SET @MySpName = 'usp_S_AddNewsAlert'
    SET @MyCustId = 10200
    SET @MyCustId = 47300

    EXECUTE @MySpName @MyCustId, @MyInstId
```

Note that in the above example there is no need for a @MyQuery variable.

4 Pitfalls

4.1 Performance

This can be poor. You must assess the usefulness of dynamic SQL vs. performance required.

4.2 Runtime parsing

Dynamic-SQL statements are generated, parsed and compiled at runtime. This means that if the statement is invalid sp_executesql will fail. It is therefore important to cater for such a situation – with proper error handling.

4.3 @MyQuery

@MyQuery should be large enough to accommodate the worst-case scenario. Typically NVARCHAR(1000) or more should be used.

4.4 CONVERT function

When converting values to NVARCHAR the conversion can be implicit or explicit. We have already seen that in some cases the latter should be used. Explicit conversion requires the use of the CONVERT function. Be aware that **CONVERT(NVARCHAR, value)** is equivalent to **CONVERT(NVARCHAR(30), value)**. This means that if **value** is longer than 30 characters then truncation will occur. Please use the correct CONVERT(NVARCHAR(xx), value) statement where xx is the maximum length of **value**.

5 Other methods

5.1 Commonly used method

Many developers would build their statement as follows:

```
...
SET @MyQuery = N'SELECT * ` +
                N'FROM ` + @TableName +
                N'WHERE CustId = ` + @MyCustId
...
```

Parameter substitutions are done inline rather than as a separate step. This method may appear more efficient but can lead quickly to unreadable code with all the added pluses '+' and commas ',' required.

5.2 Microsoft's recommended method

If you read the SQL Server documentation you will see that `sp_executesql` accepts other parameters. The first one is called `@Params` and contains the list of all parameters that have been embedded into your statement. And then following `@Params` `sp_executesql` expects one additional value for each parameter defined inside `@params`.

Example:

```
...
SET @MyQuery = N'SELECT * ` +
                N'FROM @PrmTableName ` +
                N'WHERE CustId = @PrmCustId'

SET @MyParams = N'@PrmTableName VARCHAR(100), CustId INT'

EXEC sp_executesql @MyQuery,
                  @MyParams,
                  @MyTableName,
                  @MyCustId
...
```

There are restrictions on what can and can't be done. Plus the rest of the method looks fairly complicated. You have to setup this `@MyParams` variable for a start. All variables that appear inside `@MyQuery` – i.e. anything that starts with a `@` - must appear in `@MyParams` alongside its data type. Then when calling `sp_executesql` you also have to provide values for the parameters listed in `@MyParams` in the order they appear inside `@MyParams`. All a bit messy and prone to error.