Ministerul Educatiei al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea de Calculatoare, Informatică s , i Microelectronică
Filiera Anglofonă ”Computer Science”

Dynamic SQL

Cousework on BDC

Sudent: Dariev Alexei
Verificat de: Cojanu Irina

Contents

List of Figures

List of Tables

Listings

Introduction

If you follow the various newsgroups on Microsoft SQL Server, you often see people asking why they can't do:

```
SELECT * FROM @tablename
SELECT @colname FROM tbl
SELECT * FROM tbl WHERE x IN (@list)
```

Listing 1.1 – Examples requiring dynamic sql

For all three examples you can expect someone to answer *Use dynamic SQL* and give a quick example on how to do it. Unfortunately, for all three examples above, dynamic SQL is a poor solution. On the other hand, there are situations where dynamic SQL is the best or only way to go.

In this article I will discuss the use of dynamic SQL in stored procedures and to a minor extent from client languages. To set the scene, I start with a very quick overview on application architecture for data access. I then proceed to describe the feature dynamic SQL as such, with a quick introduction followed by the gory syntax details.

1. Problem and domain analysis
1.1 Problem definition

Before I describe dynamic SQL, I like to briefly discuss the various ways you can access data from an application to give an overview of what I'll be talking about in this article.

(**Note**: all through this text I will refer to *client* as anything that accesses SQL Server from the outside. In the overall application architecture that may in fact be a middle tier or a business layer, but as that is of little interest to this article, I use *client* in the sake of brevity.)

There are two main roads to go, and then there are forks and sub-forks.

1. Send SQL statements from the client to SQL Server.
   a. Rely on SQL generated by the client API, using options like **CommandType**.**TableDirect** and methods like .**Update**.LINQ falls into this group as well.
   b. Compose the SQL strings in the client code.
      i. Build the entire SQL string with parameter values expanded.
      ii. Use parameterised queries.
2. Perform access through stored procedures.
   a. Stored procedures in T-SQL or PL SQL
      i. Use static SQL only.
      ii. Use dynamic SQL together with static SQL.
   b. Stored procedures in a CLR language such as C# or VB .Net. (SQL 2005 and later.)

Fork 1-a may be good for simple tasks, but you are likely to find that you outgrow it as the complexity of your application increases. In any case, this approach falls entirely outside the scope of this article.

Many applications are built along the principles of fork 1-b, and as long as you take the sub-fork 1-b-ii, it does not have to be bad. (Why 1-b-i is bad, is something I will come back to. Here I will just drop two keywords: SQL Injection and Query-Plan Reuse.) Nonetheless, in many shops the mandate is that you should use stored procedures. When you use stored procedures with only static SQL, users do not need direct permissions to access the tables, only permissions to execute the stored procedures, and thus you can use the stored procedure to control what users may and may not do.

The main focus for this text is sub-fork 2-a-ii. When used appropriately, dynamic SQL in stored procedures can be a powerful addition to static SQL. But some of the questions on the newsgroups leads to dynamic SQL in stored procedures that are so meaningless, that these people would be better off with fork 1-b instead.

Finally, fork 2-b, stored procedures in the CLR, is in many regards very similar to fork 1-b, since all data access from CLR procedures is through generated SQL strings, parameterised or unparameterised. If you have settled on SQL procedures for your application, there is little point in rewriting them into the CLR. However, CLR code can be a valuable supplement for tasks that are difficult to perform in T-SQL, but you yet want to perform server-side.

2. MSSQL and Oracle implementation. Particularities

A First Encounter

Understanding dynamic SQL itself is not difficult. Au contraire, it's rather too easy to use. Understanding the fine details, though, takes a little longer time. If you start out using dynamic SQL casually, you are bound to face accidents when things do not work as you have anticipated.

One of the problems listed in the introduction was how to write a stored procedure that takes a table name as its input.

Next thing to observe is that the dynamic SQL ***is not part of the stored procedure***, but constitutes ***its own scope***. Invoking a block of dynamic SQL is akin to call a nameless stored procedure created ad-hoc. This has a number of consequences:

- Within the block of dynamic SQL, you cannot access local variables (including table variables) or parameters of the calling stored procedure. But you can pass parameters – in and out – to a block of dynamic SQL if you use **sp_executesql**.
- Any USE statement in the dynamic SQL will not affect the calling stored procedure.
- Temp tables created in the dynamic SQL will not be accessible from the calling procedure since they are dropped when the dynamic SQL exits. (Compare to how temp tables created in a stored procedure go away when you exit the procedure.) The block of dynamic SQL can however access temp tables created by the calling procedure.
- If you issue a SET command in the dynamic SQL, the effect of the SET command lasts for the duration of the block of dynamic SQL only and does not affect the caller.
- The query plan for the stored procedure does not include the dynamic SQL. The block of dynamic SQL has a query plan of its own.

As you've seen there are two ways to invoke dynamic SQL, **sp_executesql** and EXEC(). **sp_executesql** was added in SQL 7, whereas EXEC() has been around since SQL 6.0. In application code, **sp_executesql** should be your choice 95% of the time for reasons that will prevail. For now I will only give two keywords: SQL Injection and Query-Plan Reuse. EXEC() is mainly useful for quick throw-away things and DBA tasks, but also comes to the rescue in SQL 2000 and SQL 7 when the SQL string exceeds 4000 characters. And, obviously, in SQL 6.5, EXEC() is the sole choice. In the next two sections we will look at these two commands in detail.

sp_executesql

sp_executesql is a built-in stored procedure that takes two pre-defined parameters and any number of user-defined parameters.

The first parameter **@stmt** is mandatory, and contains a batch of one or more SQL statements. The data type of @stmt is **ntext** in SQL 7 and SQL 2000, and **nvarchar(MAX)** in SQL 2005 and later. Beware that you must pass an **nvarchar/ntext** value (that is, a Unicode value). A **varchar** value won't do.

The second parameter **@params** is optional, but you will use it 90% of the time. @params declares the parameters that you refer to in @stmt. The syntax of @params is exactly the same as for the parameter list of a stored procedure. The parameters can have default values and they can have the OUTPUT marker. Not all parameters you declare must actually appear in the SQL string. (Whereas all variables that appear in the SQL string must be declared, either with a DECLARE inside @stmt, or in @params.) Just like @stmt, the data type of @params is **ntext** SQL 2000 and earlier and **nvarchar(MAX)** since SQL 2005.

The rest of the parameters are simply the parameters that you declared in @params, and you pass them as you pass parameters to a stored procedure, either positional or named. To get a value back from your output parameter, you must specify OUTPUT with the parameter, just like when you call a stored procedure. Note that the first two parameters, @stmt and @params, must be specified positionally. You can provide the parameter names for them, but these names are blissfully ignored.

EXEC()

EXEC() takes one parameter which is an SQL statement to execute. The parameter can be a concatenation of string variables and string literals, but cannot include calls to functions or other operators. For very simple cases, EXEC() is less hassle than **sp_executesql**. For instance, say that you want to run UPDATE STATISTICS WITH FULLSCAN on some selected tables. Here is short example of EXEC() usage:

```
DECLARE @sql varchar(1000)
DECLARE @columnList varchar(75)
DECLARE @city varchar(75)
SET @columnList = 'CustomerID, ContactName, City'
SET @city = 'London'
SELECT @sql = ' SELECT CustomerID, ContactName, City ' + ' FROM
dbo.customers WHERE 1 = 1 '

SELECT @sql = @sql + ' AND City LIKE ''' + @city + ''''
EXEC (@sql)
```

Listing 1.2 — example of usage of EXEC() statement

The EXECUTE IMMEDIATE

The EXECUTE IMMEDIATE statement executes a dynamic SQL statement or anonymous PL/SQL block. You can use it to issue SQL statements that cannot be represented directly in PL/SQL, or to build up statements where you do not know all the table names, WHERE clauses, and so on in advance. Here is a short example to find out how it works in PL\SQL:

```
execute_immediate_statement ::=
EXECUTE_IMMEDIATE dynamic_string
 {
    INTO { define_variable [, define_variable ...] | record_name }
  | BULK COLLECT INTO { collection_name [, collection_name ...] |
:host_array_name }
 }
   [ USING [ IN | OUT | IN OUT ] bind_argument
   [, [ IN | OUT | IN OUT ] bind_argument] ... ] [ returning_clause ] ;
```

Listing 1.3 — EXECUTE IMMEDIATE syntax example

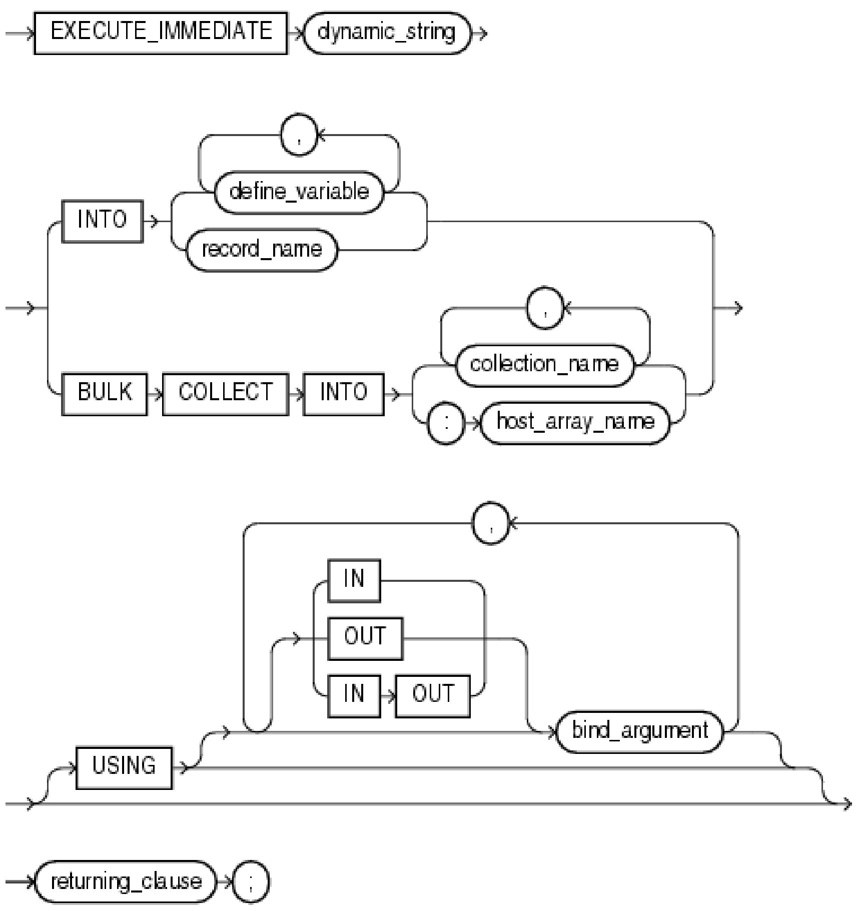

Figure 1.1 — Syntax of usage of EXECUTE_IMMEDIATE statement

MSSQL	Oracle

create procedure sql_sp_executesql
as
declare @param int
execute sp_executesql
N'select * from tab1 where
col1 = @param',
N'@param int',
@param = 35
	CREATE OR REPLACE PROCEDURE
	sql_sp_executesql AS
	v_param NUMBER(10,0);
	BEGIN
	EXECUTE IMMEDIATE 'select * from tab1
	where col1 = :1'
	USING 35;
	end;

create procedure sql_sp_executesql2
as
declare @InsOrderID int
declare @InsertString varchar(50)
SET @InsertString = N'INSERT INTO
tab1' +
' VALUES (@InsOrderID)'
EXEC sp_executesql @InsertString,
N'@InsOrderID INT',   @InsOrderID
	CREATE OR REPLACE PROCEDURE
	sql_sp_executesql2 AS
	v_InsOrderID NUMBER(10,0);
	v_InsertString VARCHAR2(50);
	BEGIN
	v_InsertString := 'INSERT INTO tab1'
	||
	' VALUES (:1)';
	EXECUTE IMMEDIATE v_InsertString
	USING v_InsOrderID;
	END;

create procedure sql_sp_executesql3
as
declare @val int
declare @InsertString varchar(50)
SET @InsertString = N'DELETE FROM
tab1 WHERE
col1 = @par1 and col2=@par2'
EXEC sp_executesql @InsertString,
N'@par1 INT, @par2 INT',   @par1=1,
@par2=@val
	CREATE OR REPLACE PROCEDURE
	sql_sp_executesql3 AS
	v_val NUMBER(10,0);
	v_InsertString VARCHAR2(50);
	BEGIN
	v_InsertString := 'DELETE FROM tab1
	WHERE col1 = :1 and
	col2=:2' ;
	EXECUTE IMMEDIATE v_InsertString
	USING 1,v_val;
	end;

Table 1.1 – Implementation in MSSQL and Oracle

Dynamic Cursors in SQL Server

The SQL Dynamic Cursors are exactly opposite to Static Cursors, and you can use this cursor to perform INSERT, DELETE, and UPDATE operations. Unlike static cursors, all the changes made in the Dynamic cursor will reflect the Original data. In this article we will show you, How to Create a Dynamic Cursor in SQL Server, and how to perform both Update, and delete operations within the dynamic cursor with example.

Dynamic Cursor in SQL Server For UPDATE

In this example we will show you, How to declare, and open a dynamic cursor in SQL Server, and how to perform the Update operation within the dynamic cursor. For this, we are using the DECLARE CURSOR Statement, and within that, we will use the WHILE LOOP to loop over the cursor elements, and perform updates.

```
SET NOCOUNT ON
-- Declaring the Variables
DECLARE @EmpID INT,
        @EmpName VARCHAR(50),
        @EmpEducation VARCHAR(50),
     @EmpOccupation VARCHAR(50),
     @EmpYearlyIncome DECIMAL (10, 2),
     @EmpSales DECIMAL (10, 2);

DECLARE dynamic_employee_cursor CURSOR
DYNAMIC FOR
     SELECT [ID]
           ,[Name]
           ,[Education]
           ,[Occupation]
           ,[YearlyIncome]
           ,[Sales]
     FROM EmployeeTable
        ORDER BY Occupation

OPEN dynamic_employee_cursor
IF @@CURSOR_ROWS > 0
BEGIN
      FETCH NEXT FROM dynamic_employee_cursor
```

```
             INTO @EmpID, @EmpName, @EmpEducation,
            @EmpOccupation, @EmpYearlyIncome, @EmpSales
      WHILE @@FETCH_STATUS = 0
      BEGIN
     IF @EmpOccupation = N'Management'
         UPDATE [EmployeeTable]
           SET [YearlyIncome] = 999999,
               [Sales] = 15000
         WHERE CURRENT OF dynamic_employee_cursor

        FETCH NEXT FROM dynamic_employee_cursor
           INTO @EmpID, @EmpName, @EmpEducation,
                @EmpOccupation, @EmpYearlyIncome, @EmpSales
      END
END
CLOSE dynamic_employee_cursor
DEALLOCATE dynamic_employee_cursor
SET NOCOUNT OFF
GO
```

Listing 1.4 — SQL server dynamic cursor for UPDATE command

Analysis

First, we used SET NOCOUNT ON stop the number of rows affected message from SQL Query. Next we declared few variables to hold the data coming from the Cursor. Then, we declared, and open the dynamic cursor called *dynamic_employee_cursor* for all the records in Employee table

Next, we used the @@CURSOR_ROWS within the IF Statement to check whether there are any rows in the Cursor or not

```
IF @@CURSOR_ROWS > 0
```

Below statement will fetch the next record from *dynamic_employee_cursor* into already declared variables.

```
FETCH NEXT FROM dynamic_employee_cursor
        INTO @EmpID, @EmpName, @EmpEducation,
             @EmpOccupation, @EmpYearlyIncome, @EmpSales
```

Then, we used the WHILE LOOP to loop over the cursor elements, and within the loop FETCH_STATUS is used to check the status of the FETCH statement.

Within the Loop, we used one more IF Statement to check whether Occupation is equal to management or not

```
IF @EmpOccupation = N'Management'
```

and if the condition is TRUE then the cursor will use the UPDATE statement to update the yearly Income, and Sales Amount.

```
 UPDATE [EmployeeTable]
 SET [YearlyIncome] = 999999,
     [Sales] = 15000
 WHERE CURRENT OF dynamic_employee_cursor
```

and then, we used the FETCH NEXT to get the next record from the cursor.

```
 FETCH NEXT FROM dynamic_employee_cursor
           INTO @EmpID, @EmpName, @EmpEducation,
                @EmpOccupation, @EmpYearlyIncome, @EmpSales
```

Lastly we used the CLOSE, and DEALLOCATE statements to close, and deallocate the cursor.

```
CLOSE dynamic_employee_cursor
DEALLOCATE dynamic_employee_cursor
```

Dynamic Cursor in SQL Server for DELETE

In our previous example we shown you, How to use the dynamic cursor for update statements. In this example we will show you, How to perform the Delete operations within the dynamic cursor.

```
 SET NOCOUNT ON
 -- Declaring the Variables
 DECLARE @EmpID INT,
         @EmpName VARCHAR(50),
         @EmpEducation VARCHAR(50),
      @EmpOccupation VARCHAR(50),
      @EmpYearlyIncome DECIMAL (10, 2),
      @EmpSales DECIMAL (10, 2);
```

```
DECLARE dynamic_employee_cursor CURSOR
DYNAMIC FOR
     SELECT [ID]
           ,[Name]
           ,[Education]
           ,[Occupation]
           ,[YearlyIncome]
           ,[Sales]
     FROM EmployeeTable
        ORDER BY Occupation, Education

OPEN dynamic_employee_cursor
IF @@CURSOR_ROWS > 0
BEGIN
      FETCH NEXT FROM dynamic_employee_cursor
            INTO @EmpID, @EmpName, @EmpEducation,
          @EmpOccupation, @EmpYearlyIncome, @EmpSales
      WHILE @@FETCH_STATUS = 0
      BEGIN
     IF @EmpOccupation = N'Management' OR @EmpEducation = N'Partial High
School'
         DELETE FROM [EmployeeTable]
         WHERE CURRENT OF dynamic_employee_cursor

        FETCH NEXT FROM dynamic_employee_cursor
           INTO @EmpID, @EmpName, @EmpEducation,
               @EmpOccupation, @EmpYearlyIncome, @EmpSales
      END
END
CLOSE dynamic_employee_cursor
DEALLOCATE dynamic_employee_cursor
SET NOCOUNT OFF
GO
```

Listing 1.5 - SQL server dynamic cursor for DELETE command

Analysis

We haven’t changed anything in this example, except few lines of code, and those lines are:

Within the Loop, we used one more IF Statement to check whether Occupation is equal to Management or Education = Partial High School

IF @EmpOccupation = N'Management' OR @EmpEducation = N'Partial High School'

and if the condition is TRUE then the cursor will use the DELETE statement to delete that record.

```
DELETE FROM [EmployeeTable]
WHERE CURRENT OF dynamic_employee_cursor
```

Dynamic cursors in Oracle

If the dynamic SQL statement is a SELECT statement that returns multiple rows, native dynamic SQL gives you the following choices:

- Use the EXECUTE IMMEDIATE statement with the BULK COLLECT INTO clause.
- Use the OPEN-FOR, FETCH, and CLOSE statements.

The SQL cursor attributes work the same way after native dynamic SQL INSERT, UPDATE, DELETE, and single-row SELECT statements as they do for their static SQL counterparts. For more information about SQL cursor attributes, see Managing Cursors in PL/SQL.

Using the OPEN-FOR, FETCH, and CLOSE Statements

If the dynamic SQL statement represents a SELECT statement that returns multiple rows, you can process it with native dynamic SQL as follows:

1. Use an OPEN-FOR statement to associate a cursor variable with the dynamic SQL statement. In the USING clause of the OPEN-FOR statement, specify a bind argument for each placeholder in the dynamic SQL statement.
2. The USING clause cannot contain the literal NULL. To work around this restriction, use an uninitialized variable where you want to use NULL.
3. Use the FETCH statement to retrieve result set rows one at a time, several at a time, or all at once.
4. Use the CLOSE statement to close the cursor variable.

```
DECLARE
  TYPE EmpCurTyp  IS REF CURSOR;
  v_emp_cursor    EmpCurTyp;
  emp_record      employees%ROWTYPE;
  v_stmt_str      VARCHAR2(200);
  v_e_job         employees.job%TYPE;
BEGIN
  -- Dynamic SQL statement with placeholder:
  v_stmt_str := 'SELECT * FROM employees WHERE job_id = :j';

  -- Open cursor & specify bind argument in USING clause:
```

```
  OPEN v_emp_cursor FOR v_stmt_str USING 'MANAGER';

  -- Fetch rows from result set one at a time:
  LOOP
    FETCH v_emp_cursor INTO emp_record;
    EXIT WHEN v_emp_cursor%NOTFOUND;
  END LOOP;

  -- Close cursor:
  CLOSE v_emp_cursor;
END;
/
```

Listing 1.6 - Using the OPEN-FOR, FETCH, and CLOSE Statements

Problems with dynamic SQL

The first major problem that needs to be tackled whenever one uses dynamic SQL commands in one's application code is SQL injection. The fact that the application accepts NVARCHAR strings and concatenates them with actual SQL queries that are later run opens up the possibility for SQL injection. Without going into the details yet just run this query for yourself (be sure to cancel the query because, as you'll notice, it loops infinitely):

```
EXEC sp_GetSalesOrderDetails @NUMBER_OF_ROWS = 2000, @COLUMN_LIST =
'''hello, your SQL injection worked'' ;
WHILE 1 = 1 SELECT ''You got owned!''; --', @UNIT_PRICE = 500
```

So, if you're familiar with SQL injections you'll notice that this dynamic SQL query is cut into two different ones by adding a ";" breaker and then running a TSQL query and cutting the rest of the query off by add comment markers "–" The query that is actually being run is the following:

```
SELECT TOP 2000 'hello, your SQL injection worked' ; WHILE 1 = 1 SELECT
'You got owned!';
-- FROM Sales.SalesOrderDetail WHERE UnitPrice >= 500
```

Granted, this SQL injection is pretty bad but it could be much worse. Depending on how wise the solution architect was the query may or may not be running with limited privileges. But even if the SQL user the application is using only has read/write privileges the person doing the SQL injection attack could replace WHILE1 = 1 SELECT'You got owned!';

with a “DELETE FROM” SQL Command. You can try it by running this query (please backup your database if you are not ready to lose the data):

```
EXEC sp_GetSalesOrderDetails @NUMBER_OF_ROWS = 2000, @COLUMN_LIST =
'''hello, your SQL
injection worked'' ; DELETE ', @UNIT_PRICE = NULL
```

This security nightmare can get a whole lot worse if an unscrupulous architect assigned a user with sysadmin privileges to the application. In that case it is not impossible to drop users, table, or worse yet, entire databases from the SQL Server instance via SQL command injection. So, please take note that one should be certain the application user has minimal security rights and try to use different data types than strings and cast them to nvarchar in the stored procedure later. This has be done in the above stored procedure for the @NUMBER_OF_ROWS and @UNIT_PRICE variables and they are therefore not at risk of SQL injection.

SQL injection is the most drastic negative point of using dynamic SQL commands but there is another weak-spot and that has to do with performance. Due to the fact that SQL Server stores optimized query plans in cache when using stored procedures one may think that using dynamic SQL in a stored procedure as above would be optimizing the system as opposed to building ad-hoc queries in the application and generating a new query plan every time they are run. However, using dynamic-SQL breaks down this capability as the original query plan becomes invalidated so the engine has to create a new one whenever a new version of the dynamic SQL command is run. You may want to look into parameterization to safely navigate those waters. Bear in mind, however, that this only applies to extreme highly transactional queries that run thousands of times per day. The caching of a new query plan is a quick process. This can be tested by used the SET STATISTICS TIME ON; command. If you run a query for the first time there is a small amount of time allocated to “parse and compile time”. This time differs between different servers with varying amounts of power. It also varies with the complexity of the query. The following query to create test data:

```
ECLARE @START_DATE DATETIME
DECLARE @ENDDATE DATETIME
SET @START_DATE = '19500101'
SET @ENDDATE = '20991231'
;
WITH CTE_DATES AS
(
SELECT
     @START_DATE DateValue UNION ALL SELECT
     DateValue + 1
FROM CTE_DATES
WHERE DateValue + 1 < @ENDDATE)

     SELECT
          CAST(DateValue AS date) AS DateTest INTO #tempTestTable
     FROM CTE_DATES
     OPTION (MAXRECURSION 0)
```

Listing 1.7 - query to create test data

Now when you run this simple query you will see that the first time you run it there is some time allocated to compilation and storing the query plan in cache:

```
SET STATISTICS TIME ON

SELECT     *
FROM dbo.#tempTestTable
WHEREDateTest BETWEEN  '19940101' AND '20050101'

SET STATISTICS TIME OFF
```

You should see some time allocated to parsing and compilation as follows:

```
SQL Server parse and compile time:
   CPU time = 69 ms, elapsed time = 69 ms.
```

However if you rerun this exact query you will notice that this time is almost non-existent:

```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 1 ms.
```

If the query test itself was changed then the engine would be forced to recompile and re-cache a new query plan. The times to do this are small but if the query is run extremely often they can add up to lots of wasted resources.

So when can one safely use dynamic SQL?
Dynamic SQL can be used by a developer to create an agile query that allows for dynamic results. However, we have seen that this practice can be very dangerous. There are, however, other uses for this type of dynamic SQL. DBAs may use it to help in maintenance/administrative tasks and indeed it works wonders for saving time. SQL injection and query plan optimization is not a problem in this case because these scripts are not run constantly and repetitively as are application/production queries.

Here is a very handy index maintenance query for any DBA that does the following using a dynamic SQL command:

1. Creates a temporary work table
2. Inserts the index names, table names and database names of all table that needs either a REBUILD or REORGANIZE (this info is included too)
3. Using cursors, the query builds a dynamic T-SQL command for each index (ALTER INDEX) and runs a rebuild on indexes over 30% fragmented and a reorganize for indexes between 10 and 30% fragmented.
4. This followed by an error handing CATCH statement so the query can keep running (this generally happens with older data-types that cannot be build ONLINE for security reasons.
5. Finally the query runs an index statistics update so the engine can use these fresh statistics for future query plans.

```
CREATE TABLE dbo.#FragTab
(DB_Name varchar(100),
[Schema] varchar(50),
[Table] varchar(200),
[Index] varchar(200),
avg_fragmentation_in_percent FLOAT,
REBUILD_Necessary BIT,
REORGANISE_Necessary BIT)

EXEC sys.sp_MSforeachdb '
USE ?;
INSERT INTO #FragTab
SELECT
DBs.name as ''DB_Name'',
dbschemas.[name] as ''Schema'',
dbtables.[name] as ''Table'',
dbindexes.[name] as ''Index'',
indexstats.avg_fragmentation_in_percent,
CASE WHEN  indexstats.avg_fragmentation_in_percent > 30 THEN 1 ELSE 0 END
as REBUILD_Necessary,
CASE WHEN  indexstats.avg_fragmentation_in_percent BETWEEN 10 AND 30 THEN
1 ELSE 0 END as REORGANISE_Necessary
FROM sys.dm_db_index_physical_stats (DB_ID(), NULL, NULL, NULL, NULL) AS
indexstats
INNER JOIN sys.tables dbtables on dbtables.[object_id] = indexstats.
[object_id]
INNER JOIN sys.schemas dbschemas on dbtables.[schema_id] = dbschemas.
[schema_id]
INNER JOIN sys.indexes AS dbindexes ON dbindexes.[object_id] = indexstats.
[object_id]
AND indexstats.index_id = dbindexes.index_id
LEFT OUTER JOIN sys.databases as DBs ON DBs.database_id =
indexstats.database_id
WHERE indexstats.database_id = DB_ID()
AND indexstats.avg_fragmentation_in_percent > 10
AND dbindexes.name IS NOT NULL
AND indexstats.database_id > 4' -- Exclude the system databases (msdb,
tempdb, master)

DECLARE @table VARCHAR(200)
DECLARE @Index VARCHAR(200)
DECLARE @DB_Name VARCHAR(100)
DECLARE @SQL NVARCHAR(4000)

DECLARE CUR_INDEXES_REBUILD CURSOR FOR SELECT
        [Table],
        [Index],
        [DB_Name]
FROM #FragTab
WHERE REBUILD_Necessary = 1

OPEN CUR_INDEXES_REBUILD

FETCH NEXT FROM
CUR_INDEXES_REBUILD INTO @table, @Index, @DB_name

WHILE @@FETCH_STATUS = 0

BEGIN
BEGIN TRY

SET @SQL = 'ALTER INDEX [' + @Index + '] ON [' + @DB_Name + '].[dbo].[' +
@table + '] REBUILD WITH (ONLINE = ON) '

EXEC sys.sp_executesql @SQL
END TRY
BEGIN CATCH
PRINT 'An error occured on the table : ' + @Table +' in the database :' +
@DB_name + ', the cursor will continue treating other indexes. The error
message is : ' + ERROR_MESSAGE()
END CATCH
```

```
FETCH NEXT FROM CUR_INDEXES_REBUILD INTO @table, @Index, @DB_name

END

CLOSE CUR_INDEXES_REBUILD
DEALLOCATE CUR_INDEXES_REBUILD


DECLARE CUR_INDEXES_REORGANIZE CURSOR FOR SELECT
        [Table],
        [Index],
        [DB_Name]
FROM #FragTab
WHERE REORGANISE_Necessary = 1

OPEN CUR_INDEXES_REORGANIZE

FETCH NEXT FROM
CUR_INDEXES_REORGANIZE INTO @table, @Index, @DB_name

WHILE @@FETCH_STATUS = 0

BEGIN

BEGIN TRY

SET @SQL = 'ALTER INDEX [' + @Index + '] ON [' + @DB_Name + '].[dbo].[' +
@table + '] REORGANIZE '

EXEC sys.sp_executesql @SQL
END TRY
BEGIN CATCH
PRINT 'An error occured on the table : ' + @Table +' in the database :' +
@DB_name + ', the cursor will continue treating other indexes. The error
message is : ' + ERROR_MESSAGE()
END CATCH

FETCH NEXT FROM CUR_INDEXES_REORGANIZE INTO @table, @Index, @DB_name

END

CLOSE CUR_INDEXES_REORGANIZE
DEALLOCATE CUR_INDEXES_REORGANIZE


DROP TABLE #FragTab

EXEC sys.sp_MSforeachdb '
USE ?;
EXEC sp_updatestats; '
```

Listing 1.8 — Example of save usage of the dynamic-SQL in T-SQL

Conclusion

In this coursework Ive briefly introducted into dynamic SQL and how to use this. It is used when we need to run the query at the moment of the runtime of the certain app. You can create more general purpose, flexible applications by using dynamic SQL because the full text of a SQL statement may be unknown at compilation. With this come the cursors: it is very usefull when we need to make a kind of pointer to a private SQL area that stores information about the processing of a SELECT or data manipulation language statement. The main problem of the dynamic SQL is that it is very vulnirable to different types of SQL injections. But it still can be usefull for developers to simplify their work. However, for maintenance or investigative reasons dynamic SQL commands can be a wonderful tool for DBAs. If we take a look from the other side this can be a good practice to try yourself in the actuall SQL injection.

References:

The Curse and Blessings of Dynamic SQL - Erland Sommarskog
PL/SQL Dynamic SQL - Oracle Documentation
Dynamic Cursor in SQL Server - By tutorialgetaway
SQL Server Commands - Dynamic SQL - Evan Barke
Using Dynamic SQL - Chalyshev M.
Dynamic T-SQL and how it can be usefeull - Anatoly Kotelevets
Execute dynamic SQL commands in SQL server - Greg Robidoux