Database Design and Development



Naming Convention, Best Practices and Guidelines

Prepared by

**Arshad Ali**

**Pradeep Venkata Satya Srikakolapu**

Consultant

Pradeep.Srikakolapu@microsoft.com

Revision and Signoff Sheet

Change Record

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Author | Version | Change Reference |
| 09/20/2013 | Arshad Ali | .1 | Initial draft for review/discussion |
|  |  |  |  |
|  |  |  |  |

Reviewers

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Version Approved | Position | Date |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table of Contents

[1 Introduction 5](#_Toc369504021)

[2 Naming Convention – Database 5](#_Toc369504022)

[3 Best Practices and Guidelines – Database 6](#_Toc369504023)

1. Introduction

The objective of this standards document is to cover all database development by providing standards as a guide. Developers must be prepared to provide a justification for any exceptions. The scope includes:

1. Database Modelling & Naming conventions
2. SQL programming
3. Transact SQL Programming Standards
4. Database Development Best Practices

Guidelines and best practices should be followed as a general rule, but it is understood that exception situations may exist.

1. Naming Convention – Database

All the objects names must follow camel case characters

1. Stored procedure – usp<Functional Name>
2. User defined function – udf<Functional Name>
3. Table - <Functional Name> , should be in singular form
4. View - <Functional Name> , should be in singular form
5. Trigger – trg<Operation><TableName>
6. Cursor - <Functional Name> , should be in singular form
7. Index – IX<Table Name><Column Name>...
8. Primary key – PK<Table Name>
9. Foreign key – FK\_<Source Table>\_<Source Key Name>\_<Target Table>\_<Target Key>
10. Default constraint – DF\_<Table Name>\_<Column Name>
11. Check constraint – CH\_<Table Name>\_<Column Name>
12. Variable - <Functional Name>
13. Parameters - <Functional Name>
14. DTS | SSIS PackagesPkg<Module>\_<Purpose>
15. Linked Server Name<DbPlatform><InstanceName><DbName>

**General guidance:**

1. Do not use reserved keywords for any object or variable names
2. For OLTP database, create schemas for each business area and place tables or views accordingly.
3. For DW database, create STG schema to keep all staging tables or views in it, DIM schema for storing dimensions and FACT schema for storing fact tables.
4. Always use schema name when referencing tables or views even though it is the default schema.
5. Never use PRIMARY filegroup for storing your user tables, create one or more separate filegroup(s) and store your user tables in that instead of PRIMARY filegroup.
6. The new filegroup should have at least 4 or 8 files equally distributed across drives (preferably with separate spindles). A single log file of the database must reside on the separate drive than the drive containing data files. If MAXDOP=4, then you should ideally create 4 files distributed across multiple physical drives with separate spindles. Please make sure log file stays away on a different drive than data files.
7. Define and implement partition strategry for larger tables especially for fact tables which contain hundreds of millions of records.

1. Best Practices and Guidelines – Database
2. **Database Design Best Practices**

Using SQL Servers RDBMS platform to its fullest potential is encouraged. Below are a few tips to aide in maximizing the built in features of relational database structures.

• Take care to normalize data as much as possible while still achieving peak performance and maintainability. Overnormalization can also be an issue as it can have a severe effect on performance and data maintenance and cause un-needed overhead. Entities and their proposed uses should be studied to formulate a data model that can balance integrity, scalability and performance. Please go thru 2nd Best practice to learn more about Normalization.

• Make use of auto-numbered surrogate keys on all tables. This can reduce data storage where compound primary and foreign keys are prevalent and simplify relationships between data when normalizing data structures and writing queries.

• Always use primary keys on tables! Even if you are enforcing integrity through other means (i.e. business layer, unique table

1. **Normalize Database structure based on 3rd Normalization form.**

**Benefits & Advantages:** [Here](http://technet.microsoft.com/en-us/library/ms191178(v=sql.105).aspx)

1. **Avoid using temporary tables & derived tables as it uses more disks I/O**

Intead use CTE(Common Table Expression); Its scope is limited to next statement in SQL Query. Using a CTE offers the advantages of improved readability and ease in maintenance of complex queries. The query can be divided into separate, simple, logical building blocks. These simple blocks can then be used to build more complex, interim CTEs until the final result set is generated. Click [here](http://technet.microsoft.com/en-us/library/ms190766(v=sql.105).aspx) to know more about CTE.

1. **Practice writing upper case for all SQL keywords.**

**Example:** SELECT, UPDATE, INSERT, WHERE, AND, OR, LIKE.

1. **There should be PRIMARY KEY in all the tables of database. The common practice is to use Primary Key as IDENTITY column.**
2. **Use SELECT statement with INSERT.**

Instead of using LOOP to insert data from TABLE 1 to TABLE 2, try to use SELECT with INSERT statement.

**INSERT INTO** Table1(Col1,Col2)

**SELECT** Col1,Col2

**FROM** Table2

**WHERE** <condition>

1. **Avoid using spaces within the names of database Objects;**

This may create issues with front end data access application. If you need spaces in your database object name then will accessing it surrounding the database object name with square brackets.

**Example:** [Product Description]

1. **Practice writing comments in stored procedures, triggers and sql batches as it won’t impact the performance.**
2. **Use Uniode datatypes such as NCHAR, NVARCHAR or NTEXT only if needed, as these datatypes take twice as much as space as non-unicode datatypes.**
3. **Always use column list in INSERT statements of SQL queries. This will avoid problem when table structure changes.**
4. **Declaring Variables**

Always declare variables at the top of your stored procedure and set their values directly after declaration. If your database runs on SQL Server 2008, you can declare and set the variable on the same line. Take a look at the following statement under SQL 2000/SQL 2005 and the second statement under SQL 2008. Standard programming language semantics are added in SQL 2008 for short assignment of values:

**DECLARE** @i int

**SET** @i = 1

**SET** @i = @i + 1

-------------------

**DECLARE** @i int = 1

**SET** @i +=1

1. **Use TRY-Catch**

To handle exception in Sql Server we have TRY..CATCH blocks. We put T-SQL statements in TRY block and to handle exception we write code in CATCH block. If there is an error in code within TRY block then the control will automatically jump to the corresponding CATCH blocks. In Sql Server, against a Try block we can have only one CATCH block.

BEGIN TRY

DECLARE @num INT, @msg varchar(200)

---- Divide by zero to generate Error

SET @num = 5/0

PRINT 'This will not execute'

END TRY

BEGIN CATCH

PRINT 'Error occured that is'

set @msg=(SELECT ERROR\_MESSAGE())

print @msg;

END CATCH

GO

Error Functions used within CATCH block

1. ERROR\_NUMBER()

This returns the error number and its value is same as for @@ERROR function.

1. ERROR\_LINE()

This returns the line number of T-SQL statement that caused error.

1. ERROR\_SEVERITY()

This returns the severity level of the error.

1. ERROR\_STATE()

This returns the state number of the error.

1. ERROR\_PROCEDURE()

This returns the name of the stored procedure or trigger where the error occurred.

1. ERROR\_MESSAGE()

This returns the full text of error message. The text includes the values supplied for any substitutable parameters, such as lengths, object names, or times.

1. **Wildcard Characters**

Try to avoid wildcard characters at the beginning of a word while searching using the LIKE keyword, as that result in an index scan, which defeats the purpose of an index. The following statement results in an index scan, while the second statement results in an index seek:

SELECT EmployeeID FROM Locations WHERE FirstName LIKE '%li' – Index Scan

SELECT EmployeeID FROM Locations WHERE FirsName LIKE 'a%i' – Index Seek

1. **ANSI-Standard Join Clauses**

Use the more readable ANSI-Standard Join clauses instead of the old style joins. With ANSI joins, the WHERE clause is used only for filtering data. Whereas with older style joins, the WHERE clause handles both the join condition and filtering data. The first of the following two queries shows the old style join, while the second one show the new ANSI join syntax:

**SELECT** a.au\_id, t.title

**FROM** titles t, authors a, titleauthor ta

**WHERE** a.au\_id = ta.au\_id **AND**

ta.title\_id = t.title\_id **AND**

t.title **LIKE** '%Computer%'

**SELECT** a.au\_id, t.title

**FROM** authors a

**INNER** **JOIN** titleauthor ta

**ON**

a.au\_id = ta.au\_id

**INNER** **JOIN** titles t

**ON**

ta.title\_id = t.title\_id **WHERE** t.title **LIKE** '%Computer%'

1. **T-SQL Variables**

Though T-SQL has no concept of constants (like the ones in the C language), variables can serve the same purpose. Using variables instead of constant values within your queries improves readability and maintainability of your code. Consider the following example:

**SELECT** OrderID, OrderDate **FROM** Orders **WHERE** OrderStatus IN (5,6)

The same query can be re-written in a mode readable form as shown below:

**DECLARE** @Order\_Delivered, @Order\_Pending

**SELECT** @ Order\_Delivered = 5, @ Order\_Pending = 6

**SELECT** OrderID, OrderDate **FROM** Orders

**WHERE** OrderStatus **IN** (@Order\_Pending, @ Order\_Pending)

1. **Offload tasks**

Offload tasks, like string manipulations, concatenations, row numbering, case conversions, type conversions etc., to the front-end applications if these operations are going to consume more CPU cycles on the database server. Also try to do basic validations in the front-end itself during data entry. This saves unnecessary network roundtrips.

1. **Check for record Existence**

If you need to verify the existence of a record in a table, don’t use **SELECT** COUNT (\*) in your Transact-SQL code to identify it, which is very inefficient and wastes server resources. Instead, use the Transact-SQL **IF EXISTS** to determine if the record in question exits, which is much more efficient. For example:

Here’s how you might use **COUNT**(\*):

IF (**SELECT** **COUNT(\*)** **FROM** table\_name **WHERE** column\_name = 'xxx')

Here’s a faster way, using **IF EXISTS**:

**IF EXISTS** (**SELECT** \* **FROM** table\_name **WHERE** column\_name = 'xxx')

The reason **IF EXISTS** is faster than **COUNT(\*)** is because the query can end immediately when the text is proven true, while **COUNT(\*)** must count go through every record, whether there is only one, or thousands, before it can be found to be true.

1. **Measure Query Performance**

**Always use statistics time feature to measure your important query and stored procedure’s performance. Use statistics time to optimize your queries Take a look at this example:**

**SET STATISTICS TIME ON**

**EXEC** <Procedure Name> <Parameters>

**SET STATISTICS TIME OFF**

The below information will be displayed in the Messages tab:

SQL Server parse and compile time:

CPU time = 6 ms, elapsed time = 6 ms.

SQL Server Execution Times:

CPU time = 24 ms, elapsed time = 768 ms.

(10 row(s) affected)

SQL Server Execution Times:

CPU time = 0 ms, elapsed time = 125 ms.

SQL Server Execution Times:

CPU time = 16 ms, elapsed time = 131 ms.

This provides a good estimation of how long the query took to be executed, showing the CPU time (processing time) and elapsed time (CPU + I/O).

1. **Use the graphical execution plan in Query Analyzer or SHOWPLAN\_TEXT or SHOWPLAN\_ALL commands to analyze your queries. Make sure your queries do an “Index seek” instead of an “Index scan” or a “Table scan.” A table scan or an index scan is a highly undesirable and should be avoided where possible especially in OLTP scenario.**
2. **You might need the length of a string variable in many places of your procedure, but don’t call the LEN function whenever it’s needed. Instead, call the LEN function once and store the result in a variable for later use.**
3. **All stored procedures must use the following template:**

SET NOCOUNT ON

SET QUOTED\_IDENTIFIER, ANSI\_NULLS, ANSI\_WARNINGS, ARITHABORT, CONCAT\_NULL\_YIELDS\_NULL, ANSI\_PADDING ON

SET NUMERIC\_ROUNDABORT OFF

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

SET XACT\_ABORT ON

**GO**

if object\_id('< procedure\_name, sysname, dbo. >') is null

EXEC(‘CREATE PROC < procedure\_name, sysname, dbo. > as SELECT 1')

**GO**

ALTER PROC < procedure\_name, sysname, dbo. >

-- TODO: supply description of the procedure

-- TODO: supply params

AS

BEGIN

--------------------------------------------

-- v1.0 Created by < author > < creation\_date, char(8), 10/10/06 >

--------------------------------------------

CODE

--------------------------------------------

END

**GO**

This template preserves existing permissions when recreating procedures and ensures that all procedures use single convenient style. When making changes that don’t mean rewriting procedure, increment minor component of the version, put new version at the beginning of the procedure and add new line in the changes section that begins with Changed by, and describes author, change date, and what kind of changes has been made. When totally rewriting procedure, increment major component of the version, do they same as above, but put Rewritten by instead of Changed by in the changes section.

If a stored procedure accepts data from external temporary table, put this table’s definition after the changes section:

/\*

**CREATE TABLE** #employee

(

employee\_id **INT**,

name **VARCHAR(128)**

)

\*/

Avoid using triggers, because they introduce “another level of indirection” to your databases. Also avoid cascading updates and deletes because this is also brings indirect logic, so use stored procedures instead.

1. **When you need to execute a string of Transact-SQL, you should use the sp\_executesql stored procedure instead of the EXECUTE statement.**

sp\_executesql gives you the possibility to use parameterised statements, EXECUTE does not. Parameterised statements gives no risk to SQL injection and also gives advantage of cached query plan. The sp\_executesql stored procedure supports parameters. So, using the sp\_executesql stored procedure instead of the EXECUTE statement improve readability of your code when there are many parameters are used. When you use thesp\_executesql stored procedure to executes a Transact-SQL statements that will be reused many times, the SQL Server query optimizer will reuse the execution plan it generates for the first execution when the change in parameter values to the statement is the only variation.

The syntax for sp\_executesql for SQL Server 2005 is

sp\_executesql [ @stmt = ] stmt

[

{, [@params=] N'@parameter\_name data\_type [ OUT | OUTPUT ][,...n]' }

{, [ @param1 = ] 'value1' [ ,...n ] }

]

The size of the string is limited only by available database server memory. On 64-bit servers, the size of the string is limited to 2 GB, the maximum size of nvarchar(max).

1. **For best performance, all objects that are called within the same stored procedure should be owned by the same object owner or schema, preferably dbo, and should also be referred to in the format of object\_owner.object\_name or schema\_owner .object\_name.**
2. **Every object should have a try..catch block with error logged into an “Error” table**

**Benefit:** Try...Catch block construct in SQL 2005/2008/2012 used for handling errors effectively instead of manually coded error handlers. Also the errors from various stored procedures of application must be logging the error details (Eg. Stored procedure name, module, error line, error message, severity) into a central location (Set of error logging tables) for better monitoring and troubleshooting.

1. **Only the outermost object participating in a nested transaction should issue a Rollback statement**

**Benefit:** Always check for the @@Trancount before rollback of a transaction. If its a nested transaction then it would rollback the parent transaction as well, hence may cause error while committing or at rollback. The code should check for the value of @@Trancount to see the count of active transactions = 1 and only rollback if the current transaction is the outermost transaction.

1. **Avoid using Count(\*) instead try below options**
   1. Query the column “ROWS” from the sys.partitions table for the relevant table for approximate count
   2. Use identify functions like DBCC CHECKIDENT() to get exact count if table has identity column and no deletes are performed
   3. Relook at the design to avoid count(\*) queries

**Benefit:** Count (\*) may be blocked if there are any active transactions on the object.

1. **Do not use functions in a query especially in a where condition**

**Benefit:** If functions are used around the column names in where clause that makes SQL optimizer not to choose the right index i.e, the expression becomes non SARGable (Searchable argument). And also using function in a query may execute the function for every row resulting in row based execution.

***For Instance***

The query lists all the invoices created in the past 10 days

SELECT InvoiceNo FROM dbo.Sales

WHERE datediff(day,BillDate ,getdate()) = 10

This query can be rewritten as (*with an index on BillDate*)

Declare @StartDate datetime

Declare @EndDate datetime

select @StartDate = getdate() - 10

select @EndDate = getdate()

SELECT InvoiceNo FROM dbo.Sales

WHERE BillDate between @StartDate and @EndDate

1. **Ensure that the columns mentioned in the ORDER BY of a SELECT statement has appropriate Indexes**

**Benefit:** Indexes with matching sort order as the ORDER BY clause provides the data from the disk in the required order avoids the in-memory sorting during runtime.

1. **Avoid creating Temporary tables in Stored Procedures that are invoked in parallel by multiple calls**

**Benefit:** Creating temporary tables (#) causes locking system pages in tempdb files before they are created. Hence too many temporary tables created in parallel by multiple calls may cause tempdb contention.

1. **Use Table Variables only if the number of rows in the variable is less than 100 rows anything beyond this should use temporary table**

**Benefit:** Table variables do not take advantage of indexes or statistics to fetch data. Use them only when the number of rows is less than 100. Table variables do not update statistics as the data is further updated/deleted/inserted. Use temporary (#) tables if the number of rows are over 100 or if the data keeps changing often.

1. **Prefer using (Select 1 From ...) instead of (Select \* From ...) when using EXISTS clause in conditional statements**

**Benefit:** Though EXISTS clause just checks the existence of a result set and comes out the moment it receives the first row “SELECT 1 FROM ...” helps improving the performance just by returning a constant (1) instead of a row/column

1. **Mention column names explicitly in the Insert statements Eg. Insert Test(Col1,Col2) Values (1,2)**

**Benefit:** This provides more clarity around what columns are being inserted with what values. Also helps when the table gets added with more columns with default values but we do not want to break the existing insert statements.

1. **“SELECT \*” should never be used. Even if every column in the table will be used for the operation, still the column names must be explicitly called out.**

**Benefit:** To eliminate columns that is not required for the operation and also provides good clarity in terms of readability.

1. **Delete statement should not be used to delete all rows from the table, instead use “TRUNCATE TABLE”**

**Benefit:** The TRUNCATE TABLE statement is a fast, minimally-logged method of deleting all rows in a table with the following advantages over Delete statement.

* Less transaction log space is used
* Fewer locks are typically used
* Without exception, zero pages are left in the table
* Identity values are reset

1. **Do not use non-Ansi joins such as “\*=” or “=\*”**

**Benefit:** These would pose problem in migration. These are legacy joins and are no longer supported from SQL 2005 onwards unless with backward compatibility.

1. **Avoid using cursors , use while loop wherever applicable**

**Benefit:** Always use 'set based approach' instead of a 'procedural approach' for accessing/ manipulating data in SQL Server. Use cursors only if it is unavoidable.

1. **Use Set Nocount On reduce network traffic**

**Benefit:** Use SET NOCOUNT ON at the beginning of your SQL batches, stored procedures and triggers in production environments, as this suppresses messages like '(1 row(s) affected)'. This in turn improves the performance of the stored procedures by reducing network traffic.

1. **Use char data type instead of varchar when the length is only 1 character (or upto 10 characters). Eg. Char(1) instead of Varchar(1)**

**Benefit:** This would avoid the overhead associated with varchar columns

1. **Do not use column numbers in ORDER BY clauses instead use the column names explicitly**

**Benefit:** changing the order of columns in the SELECT list has no impact on the ORDER BY when the columns are referred by names instead of numbers.

1. **Consider storing images/videos in file system and store only the paths in the database Vs storing images/videos directly into the database or use FileStream feature wherever applicable**

**Benefit:** Retrieving, manipulating large binary files is better performed outside the database

1. **Stored procedure must exit with proper return code. Eg. Return 0 on success and non zero in case of failure**

**Benefit:** Make sure your stored procedures always return a value indicating the status. Standardize on the return values of stored procedures for success and failures. The RETURN statement is meant for returning the execution status only, but not data. It helps caller to identify whether the procedure execution has been successful or not.

1. **Do not use GO TO control flow statement**

**Benefit:** Usage of GOTO statement makes code readability very difficult.

1. **Avoid using dynamically generated SQL Statements.**

**Benefit:** Apart from Security risk such as SQL injection, the query may also be recompiled often hence causing delayed response time and may also result in high CPU usage.

1. **Database Tuning Advisor must be run against the workload to decide up on the index requirements of the database.**

**Benefit:** Use SQL Profiler to capture the workload and provide that input to the DTA to determine what indexes would be appropriate for the system.

1. **Indexes must be created only on those columns whose selectivity of value is 5% or less.**

**Benefit:** Creating indexes on every column does not give any benefit unless they have a good selectivity.

For instance, creating an index on the column “Gender” does not offer any benefit as the possible values of this column can either only be “M” or “F”

1. **The relationship between tables must be Enforced with a foreign key**

**Benefit:** All the tables that are related to other objects must have foreign key defined in order for the query optimizer to generate optimum plan for queries

1. **The columns that have been enforced with a foreign key relationship should not allow nulls in the child table**

**Benefit:** The child table should not have nulls in the column that is related to another parent table, instead create a row with a fictious row such as “UNKNOWN”, “Not VALID” and relate it back to the child table

For Instance : An “**Order**” table may have orders without a SalesRep associated with it. However to enforce this relationship, create a Fictious row in “**SalesPerson**” table and associate this fictious row with those orders that do not have a salesrep.

1. **Persist as much data instead of calculating in query**

**Benefit:** If a query requires a calculation to be constantly performed on each row, create a computed column that is deterministic in nature and persist it to reduce impact

1. **Avoid/Minimize usage of Triggers. Promote asynchronous operations**
2. **Design Updates to a table to the minimum**

**Benefit:** Update operations should be in batches with a configurable batch size

1. **If the requirements permit, try using “NOLOCK” or read-uncommitted to reduce lock contention. If not, also look for Read-Committed Snapshot option to minimize the impact of blocking (Read : CommittedSnapshot impact on TempDB)**
2. **Do not use GUID’s as identifier or key columns**
3. **Design a sequence for all updatable objects**

In order to avoid/minimize deadlock on the server, ensure that every programmable object in SQL Server should update objects such as table in the same sequence. An easy way to do it update table in Alphabetical order.

For instance, a sales project may contain these tables

|  |
| --- |
| *Customer* |
| *Order* |
| *OrderTran* |
| *Products* |
| *Sales* |
| *SalesTran* |
| *Supplier* |

Every programmable object in SQL Server should update database objects in the same order as mentioned above

**Example**

A stored procedure to create a new invoice such as “***uspCreateInvoice***” should update “Products” table to start with followed by “Sales” and “SalesTran”

Minimize conflict resolving scenarios in occasionally connected systems design

1. **Lineage additions (DW Specific)**

* Only use surrogate keys.
* Every table should contain the columns
* IsDeleted – *default value should be FALSE*
* CreatedBy
* CreatedDateTime
* LastUpdatedBy
* LastUpdatedDateTime

1. **Join Among Tables**

* Use join columns always with the same data type attributes.
* Use whenever possible joins instead of sub-queries, since performance tends to always be better with the use of joins. Only cases where sub-queries are better are joins among linked servers.
* Whenever possible, use the ANSI SQL-92 for JOIN operations. This type of standard stipulates that the joins between tables occurs in the FROM clause and the business logic in the WHERE clause.

1. **Use of Dynamic SQL**

The use of dynamic queries should be controlled and restricted to specific situations that do not involve critical processes. The use of "dynamic SQL" requires the server to recompile the query in each execution to build a new "execution plan", making it slower, what can create problems if run multiple times and concurrently.  
Even when really need the use of "dynamic SQL", it is recommend using the procedure *sp*\_*executesql* instead of EXECUTE statement. This procedure supports parameter substitution, generating execution plans most likely to be reused. This feature makes *sp\_executesql* more efficient and versatile than EXECUTE. Furthermore, this technique reduces the risk of "SQL Injection" that can occur with the concatenation of parameters in the command mount.

1. **Recompilation**

The recompilation or not of a stored procedure is usually decided by the *Optimizer*. This rule does not apply when a stored procedure is created with the recompile parameter with which it will be recompiled each time it is executed or if it has the option KEEPFIXED PLAN, recompilation is avoided.  
In a batch process, the importance of avoiding the recompilation of a stored procedure is directly tied to the amount of times that the SP is called within the process. For example, if the SP take 2 seconds to recompile (time high), but is only called once, this may not be a serious problem. However, if it is often called within a process, this should be addressed in order to generate the lowest possible cost.  
Below there is a list the main factors that generate the recompile:

* Changes in Schema. For example, when adding or deleting an index (after the table is populated, including temporary tables).
* Threshold of change in the number of lines. The number depends on the type of the object. The following table shows these conditions:

|  |  |  |  |
| --- | --- | --- | --- |
| **Table type** | **Empty condition** | **Threshold when empty** | **Threshold when not empty** |
| Permanent | < 500 Rows | # of Changes >= 500 | # of Changes >= 500 + (20 percent of Cardinality) |
| Temporary | < 6 Rows | # of Changes >= 6 | # of Changes >= 500 + (20 percent of Cardinality) |
| Table Variable | No thresholds | No thresholds | No thresholds |

**Table 2 – Threshold number of line with changes**

* Use of some SET options:
  + - * Generally those whose query behavior or the resulting set:
      * ANSI\_DEFAULTS
      * ANSI\_NULL\_DFLT\_OFF
      * ANSI\_NULL\_DFLT\_ON
      * ANSI\_NULLS
      * ANSI\_PADDING
      * CONCAT\_NULL\_YIELDS\_NULL
      * FORCEPLAN
      * Others SET:
      * ANSI\_WARNINGS
      * ARITHABORT
      * LANGUAGE
      * NUMERIC\_ROUNDABORT
      * QUOTED\_IDENTIFIER
      * The following SET options do NOT cause recompilation:
      * ARITHIGNORE
      * CURSOR\_CLOSE\_ON\_COMMIT
      * DEADLOCK\_PRIORITY
      * FMTONLY
      * IDENTITY\_INSERT
      * IMPLICIT\_TRANSACTIONS
      * LOCK\_TIMEOUT
      * NOCOUNT
      * NOEXEC
      * PARSEONLY
      * QUERY\_GOVERNOR\_COST\_LIMIT
      * ROWCOUNT
      * STATISTICS IO
      * STATISTICS TIME
      * TRANSACTION ISOLATION LEVEL
      * XACT\_ABORT

Here are a few recommendations that should be observed during development in order to avoid unnecessary recompilations which might affect the performance of the overall of the system:

* Prefer to use the connection-level SET.
* If a stored procedure creates temporary tables, all DDL (data definition language) should be at the beginning of the SP.
* Choose table-type variables instead of temporary tables. The first does not cause recompilation. The latter should only be used for large volumes requiring temporary indexes (not supported by the first).
* For temporary tables use the KEEP PLAN hint to eliminate the threshold of 6 lines.
* Big stored procedures take longer to rebuild than small. Large SPs can be problematic, since the compilation is done at the level of SP, and not to the level of commands. Use sp\_executesql to avoid all the SP to be recompiled.
* As a last resort, identify the command that is causing the compilation and create a new stored procedure with it to avoid the work of rebuilding.

1. **Cursors**

The general rule thumb is to avoid to the maximum the use of cursors. Usually many routines implemented through cursors could be made using a best prepared "select" command. When the use of cursors is unavoidable, it is recommended using the type **FAST FORWARD**.

**FAST FORWARD**: cursor that specifies a FORWARD ONLY READ ONLY cursor with performance optimizations. Most of the time, do not use a different cursor of this.

**Important:** To speed up data processing in *loops* driven by cursors on tables with large amounts of records, it should be taken whenever possible an index associated with the column used in the where clause of the cursor. If this index is clustered, it will be even better.

1. **Views x User Defined Functions**

In addition to stored procedures, SQL Server also has User Defined Functions (UDFs). UDFs were introduced in SQL 2000 and are useful in various situations. Like any resource, they can also cause problems if misused. In general, UDFs have no performance different from other methods when called directly. However, there are situations where the misuse of functions can lead to problems, especially if they are handling large volumes of data:

* *A UDF of* ***scalar*** *type that access data is applied over a column returned by a query;*
* *A UDF of* ***scalar*** *type that access data is applied over a column on the WHERE clause of a query;*
* *A UDF of* ***table*** *type is used in a JOIN with other tables;*
* *A UDF of* ***table*** *type**is used on a column returned by a query;*
* *A UDF of* ***table*** *type**is used on a WHERE clause of a query;*

Except over some situations (UDF of type *inline table*) where the optimizer can work to produce a good access plan, the previous situations won’t provide good information to the optimizer to work leading to poor and complex access plan, bringing worse performance. *The use of UDF on those cases is not recommended. If the developer considers important its use, he must discuss it with IT Architecture team.*

Just to illustrate the previous scenario, there were projects where developers created UDFs that selected PK for processing, returning a *table data type.* These UDFs where used like tables or views on subsequent processes as part of JOIN clauses. The performance wasn’t good. In that case it’s recommended to create specific Stored Procedures that will load the PK data in a temporary work table just once. This work table could be accessed by the subsequent processes with no problem.

The following sample shows a hypothetic UDF that returns the status of a key. To resolve a query where it is necessary to put this type of result in a column, proceed like this:

Select distinct tb\_cod\_class, tb\_nom\_class,

tb\_cod\_index, tb\_nom\_index,

**dbo.fc\_titpriv\_obter\_status\_chave(T2.cod\_agrup\_gerencial)**as Cod\_Status

from tb\_tit\_privateT(nolock)

innerjoin tb\_acc\_group\_treas(nolock) T1 on ( T1.cod\_caracteristica = T.cod\_caracteristica )

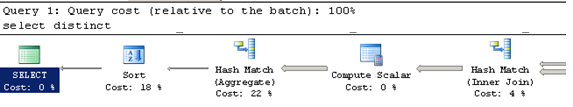
innerjoin tb\_man\_group(nolock) T2 on ( T2.cod\_agrup\_contabil = T1.cod\_agrup\_contabil )

innerjoin tb\_class(nolock) T7 on (tb\_cod\_class = T7.cod\_classificacao)

innerjoin tb\_index(nolock) t077 on (tb\_cod\_index = t077.cod\_indexador)

orderby 1, 3

The following plan occurs:



SQL Server Execution Times:

CPU time = 1609 ms, elapsed time = 1697 ms.

SQL Server Execution Times:

CPU time = 1610 ms, elapsed time = 1620 ms.

SQL Server Execution Times:

CPU time = 1546 ms, elapsed time = 1621 m

The *Hash Match* is used to allow the function to be applied to each line. The query could be written on the following alternative (with a JOIN directly over the table with the desired data):

Select distinct tb\_cod\_class, tb\_nom\_class,

tb\_cod\_index, tb\_nom\_index,

tb\_COD\_STATUS as Cod\_Status

from tb\_tit\_privateT(nolock)

innerjoin tb\_acc\_group\_treas(nolock) T1 on ( T1.cod\_caracteristica = T.cod\_caracteristica )

innerjoin tb\_man\_group(nolock) T2 on ( T2.cod\_agrup\_contabil = T1.cod\_agrup\_contabil )

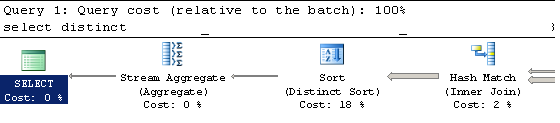
innerjoin tb\_tit\_private\_neg(nolock) T4 on ( T4.cod\_agrup\_gerencial = T2.cod\_agrup\_gerencial )

innerjoin tb\_class(nolock) T7 on (tb\_cod\_class = T7.cod\_classificacao)

innerjoin tb\_index(nolock) t077 on (tb\_cod\_index = t077.cod\_indexador)

orderby 1, 3

The following plan occurs:



SQL Server Execution Times:

CPU time = 359 ms, elapsed time = 364 ms.

SQL Server Execution Times:

CPU time = 391 ms, elapsed time = 386 ms.

SQL Server Execution Times:

CPU time = 344 ms, elapsed time = 344 ms.

With this simpler plan, note that execution plan was dropped in more than 75%

1. **Ranking Functions - Performance**

Ranking functions, introduced in SQL Server 2005, are a great enhancement to Transact-SQL. Many tasks, such as creating arrays, sequential number generation, sort results (ranking) and other similar functions, that before demanded many lines of code, can now be implemented more quickly and easily.

However, considering that critical processes and large volumes of data are typical on most of the load processing (ETL) related to DW's and DM's, reuse of these functions should be replaced by processes that bring us a better performance.

As an example there is the function ROW\_NUMBER (), which may imply a high cost in performance for tables with many records. For those cases it is recommend the use of temporary tables with one IDENTITY field. This has proven to have a much better performance in a sequential generation.

1. **Use of proper comments for modules**

Please use proper comments for your stored procedure or functions. Make sure it has enough information for others to understand the module and work on this withot any hassle in future.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

PROCEDURE NAME:

AUTHOR:

CREATED:

DESCRIPTION:

CALLED BY:

VERSION HISTORY:

DATE ALIAS Company DESCRIPTION

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/