SQL Coding Guidelines

History

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Description | Author / Editor |
| 0.1 | 28.07.2014 | Initial Version | Christian Hartl |
| 1.0 | 31.07.2014 | Final Version | Christian Hartl |
|  |  |  |  |
|  |  |  |  |

Approval

|  |  |  |
| --- | --- | --- |
| Version | Date | Approval |
| 1.0 | 31.07.2014 |  |
|  |  |  |

**Table of Contents**

1 Introduction 4

1.1 Purpose 4

1.2 Intended Audience 4

1.3 Assumptions 4

1.4 Risks 4

2 Code Commenting 5

2.1 Inline comments in code 5

2.2 File Header 5

3 Naming Standards and Formatting 6

3.1 Naming Standards 6

3.2 SQL Database Project Conventions 6

3.3 Formatting 6

4 Best Practices 8

4.1 General 8

4.2 Views 9

4.3 Stored Procedure 9

4.4 User Defined Functions (UDF) 9

4.5 Trigger 9

4.6 Usage of Data Types 9

4.7 Adding new DataTable \ DataFields 10

4.8 User Object Naming 10

4.9 SELECT \* 10

4.10 WHERE Clause 10

4.11 Consider Pagination for reports or search Stored Procedure 10

4.12 INSERT Column List 10

4.13 Use INSERT INTO instead of SELECT INTO 11

4.14 Use of Schema Qualified Tables/Views 11

4.15 Avoid Null Columns to make them mandatory and use a default value 12

4.16 NULL comparison 12

4.17 NOCOUNT option 12

4.18 Use SCOPE\_IDENTITY() instead of @@IDENTITY 12

4.19 Descriptive Names 12

4.20 Cursor usage 12

4.21 Cursor FOR UPDATE column list 13

4.22 SELECT COUNT(\*) 13

4.23 ANSI INNER Joins 13

4.24 ANSI OUTER Joins 14

4.25 Avoid unnecessary LEFT JOIN 14

4.26 Avoid unnecessary tables in JOIN 14

4.27 Explicit Index Creation 14

4.28 Primary Keys 14

4.29 UNION ALL 14

4.30 DISTINCT 14

4.31 Do not use LIKE clause when there is no Wildcard 14

4.32 Wildcard characters 14

4.33 NOT EXISTS vs. NOT IN vs. Left Joins 15

4.34 Use Derived tables 16

4.35 Temp table usage 16

4.36 Deadlocks 18

4.37 NOLOCK 18

4.38 Transactions 18

4.39 Checks 19

4.40 Use TRY… CATCH and RAISEERROR 20

4.41 Column numbers in the ORDER BY clause 20

4.42 Delete records 20

4.43 BEGIN END 21

4.44 ORDER BY 21

4.45 TOP without ORDER BY 21

4.46 Avoid Having Clause 21

4.47 Prefer IN instead of Multiple ORs 21

4.48 Use Inclusive statements 21

4.49 Prefer uncorrelated sub queries instead of correlated sub queries 22

4.50 Permission 22

4.51 Avoid Dynamic SQL 22

4.52 Dynamic SQL and SQL Injection attacks 22

4.53 Transaction Isolation Level 23

4.54 Performance 23

5 References 25

# Introduction

This document provides a set of programming and naming standards (guidelines and best practices) that apply to MS SQL code.

## Purpose

It is intended to introduce significant aspects of MS SQL. Rules and guidelines are provided to shape the development process for consistency and ease of development.

## Intended Audience

The following groups are key audiences for this document:

* Database Developers
* Application Developers

## Assumptions

The following assumptions have been made with respect to development standards

|  |  |
| --- | --- |
| Assumptions | |
|  | Developers are familiar with MS SQL 2014 |
|  |  |

## Risks

The following risks have been identified.

|  |  |
| --- | --- |
| Risks | |
|  | None. |
|  |  |
|  |  |

# Code Commenting

This section provides how different database files (stored procedure, triggers, views, etc) need to be commented and the contents that need to be incorporated during commenting. This makes the code easier to read and maintain.

## Inline comments in code

* To mark single line as comment use [--] before statement
* To mark section of code as comment use [/\* ...\*/]
* Provide inline comments where code is complex.

Explain why the code is doing this, not repeat what the code is doing

If code is no longer needed, delete it. Do not uncomment the code and leave it there

## File Header

Every stored procedure, user defined function and views must have below file header in the beginning for easy maintenance and tracing.

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

Purpose:

Created by:

Created Date:

Input Parameters

@InputParam1 -

@InputParam2 -

Output Parameters

@OutputParam1 -

@OutputParam2 -

Return Values

0 Success

1 ………………

2 ………………

Notes:

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

It is required to provide meaning full descriptions for the purpose and each parameters. Comments like “@Customner: The customer id” are not sufficient. Since stored procedure, user defined functions and views are only used in special cases, document the design decision that was made to use this and what the problem was so that regular entity framework access could not be used.

# Naming Standards and Formatting

## Naming Standards

This section describes the naming conventions that apply to different data objects of MS SQL, to make the code easier to read and maintain.

* Use the following conventions for the data objects.
  + Stored Procedure: No prefix-suffix. E.g.: GetCountries
  + UDF: No prefix-suffix. E.g.: CalculateAge
  + View: Suffix View E.g.: PlanDataView
  + Index: Suffix Index E.g.: CustomerIndex
* Do not prefix stored procedure with “sp\_” as it is reserved for system stored procedure. SQL server first tries to look for such sp in Master db. This is additional execution overhead.
* Do not use underscores.
* Do not use keywords or reserved words as object identifiers.
* Use complete English words in mixed case for table names. Do not abbreviate any part of the table name. Only use acronyms when it is generally accepted to use the acronym.
* All names including parameters must use Pascal casing
* Databases must be named in the schema of namespaces starting with the name of the software (e.g. Endeavour.Auditing)
* All tables must be placed in schemas.
* Primary keys must be named: “Id”
* Foreign keys must be named: <<ForeignTableName>>Id (Eg. PersonId)
* If multiple foreign keys to the same table are required in one table, use speaking names instead of the foreign key table name (e.g. PictureId instead of BinaryDataId)
* Constraints must be named: “<<PrimaryKeyTable>>\_<<ForeignKeyTable>>” (e.g. Person\_Address)

## SQL Database Project Conventions

This section defines the various conventions that must be followed when creating database files in the database solution in Visual Studio. In general, all databases must be created using Visual Studio and the corresponding database projects.

* Each object type (table, index, primary key, stored procedure,…) must be placed in a separate file in the project. Please be aware that the SQL for a table will automatically contain the definition for the primary key. This must be moved to a separate file
* The general structure of the database project must follow the structuring of databases as shown in SQL Server Management Studio (Tables, Security, …)
* The name of a file must contain the schema name separated with a dot from the table name
* Do not use abbreviations for objects and file names.
* The name of the primary key file must be named: <<Schema>>.<<Table>>.PrimaryKey
* Foreign key files must be named: <<Schema>>.<<PrimaryKeyTable>>.<<ForeignKeyTable>>
* Index files must be named: <<Schema>>.<<Table>>.<<IndexName>>

## Formatting

This section provides general guidelines on formatting of the SQL code, to make the code easier to read and maintain.

* Do use indentation for enhanced readability and maintenance
* To make SQL Statements more readable, start each clause on a new line and indent when needed.
* Use whitespace between words to make it more readable.
* All keywords should be in UPPERCASE
* All System Variables (@@ERROR, @@IDENTITY, etc) should be in UPPERCASE
* Pascal Casing for Table Name, Column Name and Variables.
* Do not prefix variables with @@, which should be reserved for truly global variables such as @@IDENTITY.
* Table alias should be Pascal case and should be a meaningful abbreviation instead of A, B, C
* Restrict alias name to maximum 4 characters E.g. Emp, Cust
* Use column alias for aggregate or derived columns. Casing for column alias should be Pascal Case. E.g. AVG(Salary) AS AvgSal.
* For stored procedure declare first input parameters followed by output parameters.
* Use tabs instead of white spaces and a TAB size of 4 spaces (Query Analyzer🡪Tools🡪Options🡪Editor)
* Use line breaks in SQL statements for each part of the statement. Do not write an entire SQL statement into one line; not even SELECT \* FROM table
* Ensure that the file is saved in VSS in ANSI format - This will enable us to validate the version changes by comparing 2 versions. Validation of Unicode files is difficult as they add additional space in the code after each character. In short do not use Notepad or other tools for editing the SQL file.

Example:

**Select Statement**

SELECT

Cust.CompanyName,

Ord.OrderID

FROM

Customer Cust

INNER JOIN

Order Ord

ON

Cust.CustomerID = Ord.CustomerID

**Insert Statement**

INSERT INTO dbo.SampleTable

(

Id,

Sample01,

Sample02,

Sample03

)

VALUES

(

@Id,

@Sample01,

@Sample02,

@Sample03

)

**Update Statement**

UPDATE dbo.SampleTable

SET Sample01 = @Sample01,

Sample02 = @Sample02

WHERE Id = @Id

**Delete Statement**

DELETE FROM dbo.SampleTable

WHERE Id = @Id

**Stored Procedure**

CREATE PROCEDURE dbo.InsertSampleTable

(

@Id INT,

@Sample01 DATETIME,

@Sample02 DATETIME

)

AS

BEGIN

SET NOCOUNT ON

INSERT INTO dbo.SampleTable

(

Id,

Sample01,

Sample02

)

VALUES

(

@Id,

@Sample01,

@Sample02

);

END

GO

# Best Practices

This section provides the best practices that apply to MS SQL coding so that

- Code is easier to read and debug.

- Query performs better instead of striving for the performance at the end of project.

- Better maintenance purpose.

## General

* Use DATETIME instead of DATE or TIME
* Use Unicode types for all text

## Views

* Incorporate your frequently required, complicated joins and calculations into a view so that you don't have to repeat those joins/calculations in all your queries. Instead, just select from the View.
* Don’t reference a view from another view if possible.
* Don’t use views for simple queries; use them only when creating an additional layer of abstraction makes sense. Strongly consider the use of stored procedures instead of views.
* If you need just a couple of columns from a view that returns lots of columns, consider using the base tables directly to reduce the amount of data transfer.
* Use indexed views whenever possible and needed. View are also updateable and can therefore be used like tables from entity framework.

## Stored Procedure

* Use stored procedures whenever necessary. Do not use dynamic SQL.
* Avoid intermingled DDL & DML statements they cause multiple recompiles of the SP. Recommended to place all the DDL statements at the beginning of the SP (to avoid overhead).

## User Defined Functions (UDF)

* The scalar user-defined functions (UDFs) contain one or more Transact-SQL statements that can be used to encapsulate code for reuse. Using UDFs can reduce network traffic.
* Be careful when using UDF. If the required functionality can be achieved by joining a table - join the table directly and also ensure that if paging is applied to result - Call the UDF along with TSQL only when you fetch the final result set to be shown for the page
* Store the result of an UDF in a variable and then use this variable in your sp whenever possible instead of calling the UDF multiple times.

## Trigger

* Do not use Trigger. If there is a case where trigger should be used, think again
* Triggers must be approved by the architecture team
* The trigger header must provide a detailed documentation why this trigger was implemented and what its functionality is.

## Usage of Data Types

* Always attempt to use the smallest and most appropriate data type to store the application’s numeric information. Don’t use an int when a smallint will do.
* Similarly use tinyint instead of BigInt where the value never crosses 255. For Age fields for which we know that the values are always less than 255 we can use Tinyint. This can save space and also for data we know that the length is limited and predefined - Always set the same length and don’t provide more length.
* Avoid using ntext, text and image data types as these are getting obsolete. Instead use nvarchar(max), varchar(max) and varbinary(max).
* Do not store binary or image file (BLOB) inside the database. Instead store the path to the binary or image file in the database and use that as a pointer to the actual file stored on a server.
* Use the CHAR data type for a non-nullable column instead of VARCHAR, as it will be the fixed length column.

## Adding new DataTable \ DataFields

* When adding a new field to an existing table or when creating a new table, “Description” field in the column properties should be used to describe what the field contains. Concentration should be on describing what business information is in the field \ table.
* Before adding a new table, view, or column make sure to do a thorough search of pre-existing resources through normalization. All attempts should be made to consume existing resources; especially in cases where using them can be done transparently.

## User Object Naming

sp\_: this prefix should not be used for user defined stored procedures

xp\_: this prefix should not be used for user defined extended stored procedures

fn\_: this prefix should not be used for user defined scalar functions

## SELECT \*

Use explicit fields in your select statement, do not use "SELECT \*". This technique results in fewer disks IO and less network traffic and hence better performance. Also when a table schema changes you may not get the results you expect, especially in cases where a stored procedure calls another one.

## WHERE Clause

* Use of functions in WHERE clauses should be avoided. If these columns have indexes defined on them, they become useless because of the functions. If possible the return values of the function calls should be assigned to variables, and these variables should be used in the WHERE clause. Try moving the function to the right side variable as feasible.
* When updating a single row, primary key should be specified in the WHERE clause. In case of deletion operations, carefully review deletion of all records and include condition clause (as appropriate)
* Ensure that only the required result set are returned by the query and not all the rows of the table are returned. This can reduce network traffic and boost the overall performance of the query.

## Consider Pagination for reports or search Stored Procedure

In most of the search pages, users only view selected few pages and in that case fetching all the data from the database server is not fruitful and it is waste of SQL server processing /network bandwidth application performance.

In such scenario it would be best practice to fetch only the top 10/20 records in the first execution. Using the TOP feature of SQL server would have enormous gain in SQL server performance and the benefits would be cascaded across all layers in application. So always try to implement result set pagination.

## INSERT Column List

Always use a column list in your INSERT statements. This helps in avoiding problems when the table structure changes (like adding a column).

**Not preferred code snippet without column list in Insert statement:**

INSERT INTO dbo.SampleTable

VALUES

(

1,

‘somevalue1’,

‘somevalue2’

)

**Preferred Code snippet with column list in insert statement:**

**-- Preferred**

INSERT INTO dbo.SampleTable

(

Id,

Sample01,

Sample02

)

VALUES

(

1,

‘somevalue1’,

‘somevalue2’

)

## Use INSERT INTO instead of SELECT INTO

Use ‘Insert into’ instead of ‘Select into’ to avoid locking system objects.

Do not use SELECT INTO to create your temp table, as it places locks on system objects and prevents others from executing the same query, greatly hurting concurrency and performance.

Instead, create the table using standard Transact-SQL DDL statements, and then use INSERT INTO to populate the table. This way, the system tables are not locked and multiple users will have the ability to run this same query at the same time, helping concurrency and performance

## Use of Schema Qualified Tables/Views

Prefix the table, views, etc with the server instance, db name and schema name, as this improves readability and avoids any unnecessary confusion. Qualifying table, views, etc with schema names helps in execution plan reuse.  
Server instance may not be prefixed as the server instances for development, staging and production environments may be different.

When SQL Server looks up a table/view without a schema qualification, it first searches the default schema and then the 'dbo' schema. The default schema corresponds to the current user for ad-hoc batches, and corresponds to the schema of a stored procedure when inside one. In either case, SQL Server incurs an additional runtime cost to verify schema binding of unqualified objects. Applications are more maintainable and may observe a slight performance improvement if object references are schema qualified.

**Not preferred code snippet:**

SELECT Emp.FirstName,

Emp.LastName,

Emp.Email

FROM Employee Emp

**Preferred code snippet with the proper usage of schema qualified table:**

SELECT Emp.FirstName,

Emp.LastName,

Emp.Email

FROM dbo.Employee Emp

## Avoid Null Columns to make them mandatory and use a default value

As far as possible, avoid null columns to make them mandatory and use a default value.

This will eliminate the null processing overhead.

## NULL comparison

If you need to retrieve records based on “null” values, use "is null" instead of "= null" in your where clause:

SELECT Cust.CustomerID,

Cust.CompanyName,

Cust.ContactName

FROM Customer Cust

LEFT OUTER JOIN Order Ord

ON Cust.CustomerId = Ord.CustomerID

WHERE Ord.CustomerId IS NULL

## NOCOUNT option

Explicitly mention “SET NOCOUNT OFF”. Use "SET NOCOUNT ON" only when required. When SET NOCOUNT is ON, the count (indicating the number of rows affected by a Transact-SQL statement) is not returned. This reduces the network traffic

## Use SCOPE\_IDENTITY() instead of @@IDENTITY

SCOPE\_IDENTITY() would be better to use by default than @@IDENTITY because it will give you the last used identity for the table specified in your insert statement. If you have a trigger on the table that inserts into another table with an identity, you would get that identity.

## Descriptive Names

* If you are using aliases then pick descriptive names instead of A1 or B2
* Declare your variables with descriptive names and assign either “null” or default values to them.
* Declare variables as a group for ease of reading and locating

## Cursor usage

Use cursors sparingly. Always look for alternatives to avoid cursors.

* Use a type of cursor that is sufficient for your need. If you do not need to do any updates through the cursor then consider explicitly declaring it as “READ ONLY”. Please note by default cursors are updateable.
* If you just need to scroll from first record to the last then consider declaring the cursor as “FORWARD\_ONLY” etc.
* Consider the usage of Common Table Expressions CTE instead of cursers since a CTE can fetch hierarchical and nested data very performant
* Don't use scrollable cursors unless required.
* Try to reduce the number of columns and records fetched in a cursor.
* Once you are done using a cursor, close and de-allocate your cursor:

CLOSE TitlesCursor

DEALLOCATE TitlesCursor

## Cursor FOR UPDATE column list

When a cursor is declared with a FOR UPDATE clause, it is recommended to specify a column list. SQL Server can optimize operations based on column list information available. A future release may have tighter requirements on specification of updatable columns.

## SELECT COUNT(\*)

If you need to know the total table's row count use count(1) instead.

Because SELECT COUNT(\*) statement make a full table scan to return the total table's row count, it can take lots of time for the large table.

Use “EXISTS” instead of “COUNT(\*)” when looking for the existence of one or more rows in a sub query. “EXISTS” cancels the sub query once the first existence of a record is found, while “COUNT(\*)” forces processing of the entire query. See example below;

IF (EXISTS(SELECT 1 FROM [dbo].[Employee] WHERE EmpId = 2))

## ANSI INNER Joins

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 an old style join, while the second one show the new ANSI join syntax:

**Not preferred way using Joins:**

SELECT Auth.AuId,

Ttl.Title

FROM Titles Ttl,

Authors Auth,

TitleAuthor TtlAuth

WHERE Auth.AuId = TtlAuth.AuId AND

TtlAuth.TitleId = Ttl.TitleId AND

Ttl.Title LIKE 'computer%'

**Preferred way using Joins:**

SELECT Auth.AuId,

Ttl.Title

FROM Authors Auth

INNER JOIN TitleAuthor TtlAuth

ON Auth.AuId = TtlAuth.AuId

INNER JOIN Titles Ttl

ON TtlAuth.TitleId = Ttl.TitleId

WHERE Ttl.Title LIKE 'computer%'

## ANSI OUTER Joins

Avoid non-ANSI outer joins. (\*= and =\* syntax)

Use LEFT JOIN instead of \*= and RIGHT OUTER JOIN instead of =\*.

These \* joins are not supported in SQL Server 2005

## Avoid unnecessary LEFT JOIN

A large chunk of data is retrieved by means of LEFT joins which can be an overhead.

Ensure that LEFT joins are used only when necessary. Sometimes developer tends to use LEFT join even on a not null foreign key.

Revisit db design if too many joins are used.

## Avoid unnecessary tables in JOIN

Take out unnecessary joins part of a query whose job is to return only a row count. Left joins and in some cases inner joins may be unnecessary to get the count.

In some stored procedures there is logic to first get the total row count - In that scenario ensure that the query executed does not have the left joins and udf calls as they are not really required.

## Explicit Index Creation

It is recommended that the CLUSTERED or NONCLUSTERED specifications be explicitly mentioned when creating an index. Clustered index is reserved for the primary key column of the table.

## Primary Keys

It is required that every table should have a primary key or a unique constraint as long as the table is not only used for n:m relations. Primary keys must be defined is INT. Do not use GUID as a primary key. Instead add an additional GUID column if such a functional key is needed.

## UNION ALL

Try to use UNION ALL statement instead of UNION, whenever possible.

The UNION ALL statement is much faster than UNION, because UNION ALL statement does not look for duplicate rows, and UNION statement does look for duplicate rows, whether or not they exist.

## DISTINCT

Since using the DISTINCT clause will result in some performance degradation, you should use this clause only when it is necessary.

## Do not use LIKE clause when there is no Wildcard

LIKE is generally used with wildcards.

If you use LIKE without a wildcard on a NVARCHAR column that contains Unicode data, no records will be returned even when present.

## Wildcard characters

Try to avoid wildcard characters at the beginning of a word while searching using the LIKE keyword, as those results in an index scan, which is defeating the purpose of having an index.

**The below statement with wildcard at the beginning resulting in Index Scan:**

SELECT LocationId

FROM Location

WHERE Specialities LIKE '%aples'

**The below statement resulting in the Index Seek (Preferred):**

SELECT LocationId

FROM Location

WHERE Specialities LIKE 'a%s'

## NOT EXISTS vs. NOT IN vs. Left Joins

Explore different ways of achieving the same results and see which performs best for your scenario.

Below is an example which shows 3 different ways of getting the same results. The query pertains to the common scenario of finding parent table rows that don’t have a match in the child table.

**Below code snippet with “NOT IN”:**

SELECT Cust.CustomerID,

Cust.CompanyName,

Cust.ContactName

FROM Customer Cust

WHERE Cust.CustomerId NOT IN

(SELECT DISTINCT Ord.CustomerId

FROM Order Ord)

**Below code snippet with “LEFT JOIN”:**

SELECT Cust.CustomerID,

Cust.CompanyName,

Cust.ContactName

FROM Customer Cust

LEFT OUTER JOIN Order Ord

ON Cust.CustomerId = Ord.CustomerID

WHERE Ord.CustomerId IS NULL

**Below code snippet with “NOT EXISTS”:**

SELECT Cust.CustomerID,

Cust.CompanyName,

Cust.ContactName

FROM Customer Cust

WHERE NOT EXISTS

(SELECT Ord.CustomerId

FROM Order Ord

WHERE Cust.CustomerId = Ord.CustomerId)

In each case, the above query will return identical results. But, which of these three variations of the same query produces the best performance? Assuming everything else is equal; the worst performing version through the best performing version will be from top to bottom, as displayed above. In other words, the NOT EXISTS variation of this query is generally the most efficient.

We say generally, because the indexes found on the tables, along with the number of rows in each table, can influence the results. If you are not sure which variation to try yourself, you can try them all and see which produces the best results in your particular circumstances.

## Use Derived tables

Use 'Derived tables' wherever possible, as they perform better. Consider the following query to find the second highest salary from Employees table:

SELECT MIN(Salary)

FROM Employee

WHERE EmpId IN (SELECT TOP 2 EmpId

FROM Employee

ORDER BY Salary DESC)

The same query can be re-written using a derived table as shown below, and it performs twice as fast as the above query:

SELECT MIN(Salary)

FROM (SELECT TOP 2 Salary

FROM Employee

ORDER BY Salary DESC)

## Temp table usage

Avoid the creation of temporary tables while processing data, as much as possible, as creating a temporary table means more disks IO. Consider advanced SQL or views or table variables of SQL Server 2000 or derived tables, instead of temporary tables. Table variables used in stored procedures result in fewer recompilations of the stored procedures than when temporary tables are used. Table variables are cleaned up automatically at the end of the function, stored procedure, or batch in which they are defined. Keep in mind that, in some cases, using a temporary table performs better than a highly complicated query:

**Below code snippet using Temp table:**

CREATE PROC GetProductStats AS

BEGIN

CREATE TABLE #OrderCounts

(

ProductID INT,

OrderCount INT

)

INSERT INTO #OrderCounts

SELECT ProductID,

COUNT(DISTINCT OrderID)

FROM [Order Details]

GROUP BY ProductID

ORDER BY COUNT(DISTINCT OrderID) DESC

SELECT COUNT(ProductID),

OrderCount

FROM #OrderCounts

GROUP BY OrderCount

ORDER BY OrderCount DESC

END

**Below code snippet using table variable instead of temp table:**

CREATE PROC GetProductStats AS

BEGIN

DECLARE @OrderCounts TABLE(ProductID INT,

OrderCount INT)

INSERT INTO @OrderCounts

SELECT ProductID,

COUNT(DISTINCT OrderID)

FROM [Order Details]

GROUP BY ProductID

ORDER BY COUNT(DISTINCT OrderID) DESC

SELECT COUNT(ProductID),

OrderCount

FROM @OrderCounts

GROUP BY OrderCount

ORDER BY OrderCount DESC

END

**Guidelines for temp table vs. table variables**

* Stored procedures that use table variables won't need to be recompiled as often as stored

procedures that use temporary tables. This speeds up your stored procedures.

* If you need to work with a large amount of data with many indexes, use a temporary table

instead of a table variable. You can't apply the CREATE INDEX statement to a table variable.

SQL Server will still automatically create indexes when you use a UNIQUE or PRIMARY KEY

constraint on a table variable, so if you only need unique indexes you need not worry about this

issue.

* Create indexes on temp table if the table holds a large result set. The index has to be created first

before data is dumped into the table. Ensure that the index is used on column which is going to

be used in the primarily joining column of the temp table. Best option is to create clustered index.

Ensure that field on which index is created is unique index

* You can't apply the ALTER TABLE statement to a table variable. If your working table needs to

change its schema in the course of a batch, use a temporary table instead.

* You can't insert the results of a stored procedure into a table variable, or make one the target of a

SELECT INTO statement.

* As a general rule of thumb, you should use table variables any time that your temporary data is of

reasonable size and is only used a few times. As your temporary data grows in size, complexity,

and reuse, a temporary table will be more appropriate. **If number of records estimated to be**

**stored are less than 100 then use table variables else use Temp tables**

* Of course, you don't have to make this decision blindly. With execution plans, SQL Profiler, and

performance counters available to monitor what's going on inside of your stored procedures, you

can code the alternatives and test them when there's any doubt.

* One factor that you should not take into account is the trade-off between disk space and RAM

usage. Many references on the Web claim that table variables are implemented purely in

memory, as opposed to temporary tables which are actually written to a physical database. In

fact, both table variables and temporary tables are created in the tempdb database. After all, SQL

Server doesn't know how much data you're going to try to put into a table variable; what if you

want to put in more than you have RAM available? In any case, SQL Server will cache small

amounts of data in RAM no matter which means you use to declare the table.

## Deadlocks

All the stored procedure and triggers should always consistently access tables in the same order.

This helps in avoiding deadlocks.

Other things to keep in mind to avoid deadlocks are:

* Keep your transactions as short as possible.
* Touch as less data as possible during a transaction.
* Never, ever wait for user input in the middle of a transaction.
* Do not use higher level locking hints or restrictive isolation levels unless they are absolutely needed.

## NOLOCK

In case of Search or Report stored procedures, use individual WITH (NOLOCK) table lock hints for querying table that gets frequently locked. This can reduce locking and blocking.

Do not use WITH NOLOCK hints in sps that modify data.  
Alternatively, refer the section Transaction Isolation Level

## Transactions

* Transactions should be enclosed in BEGIN TRANSACTION code and end in a COMMIT

TRANSACTION or ROLLBACK TRANSACTION statement. Data manipulations should be

contained within transactions.

BEGIN TRANSACTION;

BEGIN TRY

    -- Generate a constraint violation error.

DELETE FROM Production.Product

WHERE ProductID = 980;

END TRY

BEGIN CATCH

    SELECT

ERROR\_NUMBER() AS ErrorNumber

,ERROR\_SEVERITY() AS ErrorSeverity

,ERROR\_STATE() AS ErrorState

,ERROR\_PROCEDURE() AS ErrorProcedure

,ERROR\_LINE() AS ErrorLine

,ERROR\_MESSAGE() AS ErrorMessage;

IF @@TRANCOUNT > 0

ROLLBACK TRANSACTION;

END CATCH;

IF @@TRANCOUNT > 0

COMMIT TRANSACTION;

GO

* Although every 'BEGIN TRANS' increments @@TRANCOUNT by one and every 'COMMIT

TRANS' decrements the count, a 'ROLLBACK TRANS' rolls back all the changes. In other words, a rollback sets @@TRANCOUNT all the way back to zero.

So in nested sps with transactions, attempting to execute a second rollback generates an error

with the message 'The ROLLBACK TRANSACTION request has no corresponding BEGIN

TRANSACTION.'

Hence check for @@TRANCOUNT value before starting, committing or rolling back any transaction.

* Multiple transaction statements should use common error handling code.

## Checks

* Check the existence of DB object before creating it:

IF EXISTS

(

SELECT Name

FROM dbo.sysobjects

WHERE Id = object\_id(N'[dbo].[GetData]')

AND OBJECTPROPERTY(id, N'IsProcedure') = 1

)

DROP PROC

* In case of updating, we should restrict the value updated to only those rows for which the value is intended

For e.g.: If we want to set the Supervisor ='B' in Employee for City = 'Chn'

It is beneficial to write the query:

Update Employee

Set Supervisor = 'B'

Where City = 'Chn' AND

Supervisor <> 'B'

to avoid no of rows being updated and also reducing deadlocks as most deadlocks occur due to update statements and it is better to keep range of updates to bare minimum.

## Use TRY… CATCH and RAISEERROR

Use the TRY…CATCH constructs instead of checking the global variable @@ERROR. Because the @@ERROR variable value is reset after each SQL statement, this antiquated approach leads to rather bloated stored procedures, as the variable must be checked after each statement with code to handle any problems. The TRY...CATCH block in SQL Server 2005 offers a much more readable syntax and one that developers are more familiar with.

Avoiding @@ERROR also prevents usage of GOTO statement (which often creates problems).

These TRY..CATCH are not supported in SQL Server 2000 where @@ERROR is preferred.

BEGIN TRY  
DECLARE @MyInt INT;  
-- Following statement will create Divide by Zero Error  
SET @MyInt = 1/0;  
END TRY  
BEGIN CATCH  
DECLARE @ErrorMessage NVARCHAR(4000);  
SELECT @ErrorMessage = ERROR\_MESSAGE();  
RAISERROR (@ErrorMessage, 16, 1);  
END CATCH;  
GO

## Column numbers in the ORDER BY clause

Do not use the column numbers in the ORDER BY clause as it impairs the readability of the SQL statement. Further, 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. Consider the following example, in which the second query is more readable than the first one:

**Not preferred code:**

SELECT OrderId,

OrderDate

FROM Order

ORDER BY

2

**Preferred code:**

SELECT Orderid,

Orderdate

FROM Order

ORDER BY

OrderDate

## Delete records

If you need to delete all records in a table, try truncating instead of deleting.  
 “TRUNCATE TABLE” removes all rows from a table without logging the individual row deletes.

TRUNCATE statement requires the user to have privileges that a runtime service account may not have. So on a case by case basis see if this statement can be availed or not.

TRUNCATE TABLE Author

## BEGIN END

Always use the BEGIN and END statements anywhere a control-of-flow statement must execute a block of one or more Transact-SQL statements. The BEGIN and END statements are used to group multiple Transact-SQL statements into a logical block:

BEGIN

BEGIN TRY  
 DECLARE @MyInt INT;  
 -- Following statement will create Devide by Zero Error  
 SET @MyInt = 1/0;  
 END TRY  
 BEGIN CATCH  
 DECLARE @ErrorMessage NVARCHAR(4000);  
 SELECT @ErrorMessage = ERROR\_MESSAGE();  
 RAISERROR (@ErrorMessage, 16, 1);  
 END CATCH;  
END

GO

## ORDER BY

Don't use ORDER BY in your SELECT statements unless you really need to, as it adds a lot of extra overhead.

## TOP without ORDER BY

It is generally recommended to use specify ORDER BY clause when using TOP. Otherwise, results will be plan dependent.

## Avoid Having Clause

The HAVING clause is used to restrict the result set returned by the GROUP BY clause. When you use GROUP BY with the HAVING clause, the GROUP BY clause divides the rows into sets of grouped rows and aggregates their values, and then the HAVING clause eliminates undesired aggregated groups. In many cases, you can write your Select statement so that it will contain only WHERE and GROUP BY clauses without HAVING clause. This can improve the performance of your query.

## Prefer IN instead of Multiple ORs

Keep the amount of code to a minimum. Consider below query:

SELECT \* FROM CodeDetail

WHERE CategoryNbr = 1

OR CategoryNbr = 2

OR CategoryNbr = 3

You can get the same results with less typing if you use IN.

SELECT \* FROM CodeDetail

WHERE CategoryNbr IN (1,2,3)

## Use Inclusive statements

Avoid negative search conditions, such as “NOT IN”, “NOT EXISTS”, “<>”, etc. Indexes can rarely utilize an operation with a “NOT” properly, therefore, these clauses tend to be costly operations in processing and I/O. Re-work the query to handle the same expression without using the “NOT” operation. Minimize the use of not equal operations, “<>” or “!=”. SQL Server has to scan a table or to find all values to see if they are not equal to the value given in the expression. Try rephrasing the expression using ranges.

Use =, IN and BETWEEN which are inclusive statements.

Use Between instead of >= and <=.

## Prefer uncorrelated sub queries instead of correlated sub queries

Use uncorrelated sub queries instead of correlated sub queries. Uncorrelated sub queries are those where the inner SELECT statement does not rely on the outer SELECT statement for information. In uncorrelated sub queries, the inner query is run once instead of being run for each row returned by the outer query.

## Permission

* No tables should have Insert, Update, Delete permissions. These can only be done through

stored procedures.

* Creates an entry in the security system that allows a user in the current database to work with

data in the current database or execute specific Transact-SQL statements.

## Avoid Dynamic SQL

Avoid dynamic SQL statements as much as possible. Dynamic SQL tends to be slower than static SQL, as SQL Server must generate an execution plan every time at runtime. IF and CASE statements come in handy to avoid dynamic SQL. Another major disadvantage of using dynamic SQL is that, it requires the users to have direct access permissions on all accessed objects like tables and views. Generally, users are given access to the stored procedures which reference the tables, but not directly on the tables. In this case, dynamic SQL will not work.

If at all Dynamic SQL should be used, then use sp\_executesql to run the dynamic SQL. Parameterisation helps in plan reuse.

## Dynamic SQL and SQL Injection attacks

While building dynamic SQL statements, ensure that the code is not vulnerable to SQL Injection attacks. Use sp\_executesql and parameterised inline queries if any.

Consider the example in which Dynamic SQL is written to fetch the Order Details as per the ship name entered by the naïve users from front end. In that case user can execute the malicious code by entering the ‘DROP TABLE ORDER – ‘as shown in below code snippet.

SELECT OrderID,

OrderDate,

CustomerID,

ShipName

FROM dbo.Order

WHERE 1 = 1 AND

ShipName LIKE ‘’ DROP TABLE ORDER – ‘

If the user has the appropriate rights for dropping the table, then user would be successful in attacking through SQL Injection.

In order to prevent this, parameterised statement and sp\_executesql should be used as shown in below code snippet:

CREATE PROCEDURE SearchOrders

@CustId NCHAR(5) = NULL,

@ShipName NVARCHAR(40) = NULL AS

DECLARE @Sql nvarchar(4000)

SELECT @Sql = ' SELECT OrderID, OrderDate, CustomerID, ShipName ' +

' FROM dbo.Order WHERE 1 = 1 '

IF @CustId IS NOT NULL

SELECT @Sql = @Sql + ‘ AND CustomerID LIKE @CustID ‘

IF @ShipName IS NOT NULL

SELECT @Sql = @Sql + ‘ AND ShipName Like @ShipName ‘

EXEC sp\_executesql @Sql, N’@CustId NCHAR(5), @ShipName VARCHAR(40)’, @custId, @ShipName

Since the SQL String does not include any user input, there is no opening for SQL Injection.

## Transaction Isolation Level

In case of Search and Report stored procedures - where the query is intended to just provide the report and not going to do any updates, Transaction Isolation Level Read Uncommitted should be used.

Note: This should not be set for stored procedures which perform any insert/update or delete and have any transactions set in it

By setting the transaction isolation level as Read Uncommitted, the search and report stored procedure will not face any blocking due to any locking of tables.

Syntax:

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

## Performance

* Check whether relevant indexes exist for the query
* Check if the query plan for the query is appropriate or it indicates anomalies
* Check if the query is being tested with the right amount of data
* Check if parameter values being used for the stored procedure would influence the execution and if the indexes are appropriate for the likely combinations
* Check if a query needs included indexes (covered queries)
* Check if the query is using any hints at all and if so whether the hints are right (this includes index hints, join hints, isolation/lock hints etc)
* Check if it is faster to create XML using SQL XML instead of creating an XML from a dataset using loops
* Check if the application is missing valid filter criteria in the query and is attempting to filter in the code
* Check whether for the query, using stored procedures with output parameters would be a more appropriate choice rather than returning a result set
* Check if the clustered index is appropriately and judiciously chosen as there can be only one clustered index in a table. Check if the primary key must be the clustered index too.
* Check the time taken by the query and record the same and match against expectation
* Check the size of the columns used in indexes
* Ensure that the query does not miss relevant join predicates
* Check whether appropriate indexes exist for the join conditions
* Follow guidelines on writing better queries for linked server
* Check if the intended query should be executed online or offline via a job
* Check if the query should be executed in the OLTP or reporting system
* Check if Composite Clustered index keys are kept as small as possible.
* Consider creating indexes on columns used in WHERE, ORDER BY, GROUP BY, and DISTINCT clauses.
* Unused Indexes should be removed.
* Avoid table scans and index scans in queries in SP. Ensure that index seek is shown in query in Actual Execution plan.

We can use tools such as SQL Profiler, Database Tuning Advisor (DTA) and Execution Plan through Management Studio to verify the expected performance. Similarly, SQL DMV tool from MS SQL Server 2005 helps in verifying the performance. Refer books and online material for details on usage of these tools as it is out of scope for this document.

# References

[1]

[2]

[3]

[4]

[5]

[6]

[7]

[8]

[9]