Best Practices and Coding Standards

For Microsoft SQL Server

Document History

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Date** | **Version** | **Revision Details** | **Author(s)** | **Reviewer** | **Approver** |
| 06-24-2013 | 1.0 | Initial Draft | Mahesh KM |  |  |
| 07-22-2013 | 1.01 | Added script rules | Mahesh KM |  |  |
|  |  |  |  |  |  |

Table of Contents

[1. Summary of Purpose 4](#_Toc227489165)

[1.1. Major Areas 4](#_Toc227489167)

[2. Code Usage 4](#_Toc227489168)

[2.1. Stored Procedures 5](#_Toc227489169)

[2.2. Functions 6](#_Toc227489170)

[2.3. Views 6](#_Toc227489171)

[2.4. Triggers 6](#_Toc227489172)

[3. Code Style Guide 6](#_Toc227489173)

[3.1. Naming requirements 6](#_Toc227489174)

[3.2. Documentation 7](#_Toc227489175)

[3.2.1. Code object header 7](#_Toc227489176)

[3.2.2. Inline documentation 8](#_Toc227489177)

[3.3. ANSI and SQL Settings 9](#_Toc227489178)

[3.3.1. ANSI settings 9](#_Toc227489179)

[3.3.2. In Code settings 9](#_Toc227489180)

[3.4. Code formatting 9](#_Toc227489181)

[3.4.1. Indention 10](#_Toc227489182)

[3.4.2. Capitalization 11](#_Toc227489183)

[3.4.3. Parameterization 11](#_Toc227489184)

[4. Development best practices 11](#_Toc227489185)

[4.1. Query processing…………………………………………………………………………………………………………………….12](#_Toc227489186)

[4.1.1. Record set (SELECT clause) 12](#_Toc227489187)

[4.1.2. Joining Tables (FROM clause) 12](#_Toc227489188)

[4.1.3. Filtering Data (WHERE Clause) 12](#_Toc227489189)

[4.1.4. Grouping Data (GROUP BY Clause) 12](#_Toc227489190)

[4.1.5. Ordering data (ORDER BY Clause) 12](#_Toc227489191)

[4.2. Procedural logic 13](#_Toc227489192)

[4.2.1. Cursors 13](#_Toc227489193)

[4.2.2. WHILE Loops 13](#_Toc227489194)

[4.2.3. IF statements 15](#_Toc227489195)

[4.2.4. GOTO statements………………………………………………………………………………………………………….16](#_Toc227489196)

[*4.3 Subqueries, Temporary tables, CTE’s, and Unions*……………………………………………………………………………..16](#_Toc227489197)

5. File Organization/Versioning……………………………………………………………………………………………………………...19 6 . Summary …………………………………………………………………………………………………………………………………... . 19 7 …Check List………………………………………………………………………………………………………………………………………… 24

Summary of Purpose

This document specifies styles and best practices used for developing database code in Microsoft SQL 2008 and later, including; stored procedures, user defined functions, views, and triggers. It is anticipated that this document will be used as a guide during development of database code objects.

## Major Areas

The document will outline controls and usage patterns for database code objects .

The style guide will outline controls in the following areas:

* Guidelines for using the different types of database code objects
* Documentation expected in code objects; both internal and external
* Proper indention and capitalization to be used in code objects.
* Template information to be used for documents.

The best practices section outlines usage patterns in the following area:

* Parameterization guidelines in stored procedures and functions.
* Common Table expressions(CTE), temporary tables and table variable tables
* Cursor usage
* Unions, sub queries
* Optimization of database code.

# Code Usage

Database code objects exist to support and enforce the referential and data integrity of an RDBMS. For MsSQL there are four types of databases code objects used

1. Stored Procedures
2. Functions
3. Views
4. Triggers

For all code objects the following criteria should be met:

* Code should be optimized for performance by properly utilizing database features such as indexes and parallelism.
* Transactions should remain as short as possible and should minimize blocking naturally without the use of table hints.
* Database code should be limited to CRUD functions and should not contain business logic that could change over time.
* Database code should be generic in nature and should support reuse as much as possible.
* Each type of database code has specific uses and usage patterns.

## Stored Procedures

The following guidelines should be used when designing Stored Procedures.

* Dynamic SQL should be avoided in stored procedures as this can cause performance issues and introduce SQL injection points into the code.
* Parameter sniffing should be avoided. Eg: [Click Here](https://www.simple-talk.com/sql/t-sql-programming/parameter-sniffing/)
* Stored procedures should always be locally scoped. SCOPE\_IDENTIY() is preferred to @@IDENTITY and global stored procedure should not be used.
* Minimize the use of Cursor
* Add transactions and error handling ( try catch )
* Add SET NO COUNT ON in procedures
* Avoid using sub queries . replace it with JOIN
* Avoid Select \*, Use Column List

## 2.2 Functions

Functions are used to store code as a reusable object in the database. Generally functions contain business logic, and therefore should be used sparingly. The following guidelines should be used when designing Functions:

* The logic of a function should have no exceptions or change over time.
* Scalar functions should be self contained. Functions that retrieve data from tables within the database should be limited to table functions.
* Functions should always be deterministic (Return the same result giving the same parameters and database state)

## 2.3 Views

Views act as virtual tables and should be used for standardizing common data access patterns within a data model. Views have the following design goals:

* Hide the complexity of the underlying database schema
* Control access to rows and columns of data.
* Aggregate data for performance.
* All objects referenced in the view must use two part naming (<Scehma>.<Object\_name>)
* If a view uses an aggregate function the row set must include a COUNT\_BIG(\*) column.

## 2.4Triggers

The primary goal of triggers is to support the referential integrity of a database by cascading data changes throughout the database. With the introduction of cascading as a function of a foreign key constraint in the ANSI: SQL-92 standard, the use of triggers has a narrower focus. There are some standard usage patterns that still use triggers though; such as auditing data, and managing complex referential integrity tasks. When designing triggers there are two primary caveats to watch out for; Long transactions, and business logic.

* *Long transactions* come in the form of cascading or nesting. Both cascading and nesting happen a long series of triggers are fired one by the other. The difference is cascading involves triggers on different tables where nesting involves triggers on the same table.
* *Business logic* generally has exceptions and will change over time as the business changes. Since the goal of a trigger is to maintain referential integrity, it would be difficult to maintain historical accuracy when the logic of the trigger changes.

# Code Style Guide

Database code objects are synonymous with procedures and functions in the application and therefore subject to change management In general, the guidelines for writing database code creation scripts are as follows:

* The script should check for an existing code object matching the name and delete it if it exists
* ANSI settings should not be set in a script. These settings should be left to the connection and server defaults.
* While MsSQL has ALTER commands for all code objects, the object will always be scripted with the CREATE command.

## Naming requirements

Use camel case, with the first letter of words putting up like camel humps.

Use plural names for tables and views because they are actually containers of a group of same type rows.

* **Table**.
* Pascal notation; end with an ‘s’
* MeaningFull name start with capital letter
* Example : Book Table – Books
* **View**
* Use “VW” As prefix.
* “VW” + MeaningFull name start with capital letter
* Example : Book View – VWBooks
* **User Defined Functions**
* Use “fn” as prefix.
* “fn” + MeaningFull name start with capital letter
* Example : Get Book Name Function – fnGetBookName
* **Stored Procedures**
* Do not use sp\_ as a prefix
* One of the things you do not want to use as a standard is "sp\_".  This is a standard naming convention that is used in the master database.  If you do not specify the database where the object is, SQL Server will first search the master database to see if the object exists there and then it will search the user database. So avoid using this as a naming convention.
* use “usp\_” as prefix.
* Example : usp\_ViewGroupData
* ***Columns***
* Use Pascal standards
* Example : FirstName
* **Trigger**
  + TR\_<TableName>\_<action>
  + Use “TR” as Prefix
  + Examples: TR\_Orders\_UpdateProducts
  + Notes: The use of triggers is discouraged

**Primary Key**

Try to create the primary key column with integer data type and Clustered Index

* Name should be PK\_TableName\_ColumnName
* **Foreign Key**
* Name Should be FK\_TableName\_ColumnName
* **Check Key**
* Name should be CK\_TableName\_ColumnName
* **Default**
* Name should be DF\_TableName\_ColumnName

## Documentation

Documentation can be done inside of a code object and outside of a code object in the form of a header. Unless otherwise determined by the data management group, documentation should always be in English.

### Code object header

Each object should contain a header as part of the script. The header should be the first statement before the CREATE statement to ensure it is included as part of the object definition in MsSQL. Object header templates can be found in Appendix 1 and will include some of the following categories

* **Name:** The fully qualified three part name of a procedure, this not only defined the procedure but identifies the data model and schema it belongs to. This category will exists in all headers
* **Description:** A short description of the purpose and function of the object. This category will exists in all headers
* **Input Parameters:** Lists the name data type and description of each input parameter. When applicable, the description should contain the table and column that the parameter will reference. This category will only exist on stored procedures and functions.
* **Output Parameters:** List the name, data type and description of any declared output parameters or the result of a scalar function. When applicable, the description should contain the table and column name the parameter references. It is not necessary to list complete records sets returned by the object. This category will only exist on stored procedures and functions.
* **Author Legend:** List the initials and name of everyone making modifications to the procedure. This category will exists on all object headers
* **Modification History:** List the initials, change date, and description of all modifications done to an object. Initials should match the Author legend.

### Inline documentation

*“Good code is its own best documentation. As you're about to add a comment, ask yourself, 'How can I improve the code so that this comment isn't needed?' Improve the code and then document it to make it even clearer. “*

Steve McConnell ***Code Complete***

For the most part SQL is already self documenting and should not need comments other than the object header. However, there will be an occasion where comments will be necessary to the understanding of the intent of a SQL statement. The following guidelines should be used when documenting SQL objects.

* Exceptions and assertions being made in the code should be well documented either in the object header or in-line in large objects.
* Variables declared in the code should be documented as to their intent.

See the following example:

IFEXISTS(SELECT 1 FROMsys.ObjectsWHERE name=N'usp\_ViewGoupData'ANDtype=N'P')

DROPPROCdbo.usp\_ViewDRoupData

GO

/\*=============================================================

//

// Name: usp\_ViewGoupData

//

// Description:

// Gets the group member details...

//

//

// Input Parameters:

// @AgencyId INT - Agency Id

// @GroupId INT - Group Id

//

// Output Parameters:

// Record set containing

// Code - Varchar(6)

// GroupName - Varchar(20)

//

// Notes:

// This is a sample procedure used by the Sp Style guide only

//

//

// Author Legend:

// Initials Name

// MKM Mahesh KM

//

// Maintenance Log:

// Who When What

// --- ---------- ----------------------------------------------

// MKM 24/06/2013 Procedure created

//

//=============================================================\*/

CreateProcedure dbo.usp\_ViewGoupData

(

@AgencyId INT

,@GroupId INT

)

AS

BEGIN

SETNOCOUNTON

SELECT

G.Code

,G.Name AS GroupName

FROM

dbo.Usergroup G

INNERJOIN dbo.Agency A

ON A.Agency\_id = G.AgencyId

WHERE

A.Agency\_id = @AgencyId

AND G.Group\_Id =@GroupId

GO

## ANSI and SQL Settings

### ANSI settings

In the past, ANSI settings have been set as part of the creation of an object. However in future release of MsSQL, all ANSI settings will be forced on, so it is not necessary to include these in code object scripts.

### In Code settings

In general, SQL connection settings should be left at the server defaults. The following exceptions apply.

* **SET NOCOUNT ON** – This setting id desirable in stored procedure to eliminate the send row set containing the count of records that where returned.
* **SET ARITHABORT ON|OFF** – In some cases is desirable to turn off the ARITHABORT to avoid divide by 0 errors. In cases where this is specified, it should be the first statement in the object’s declaration.
* **SET ROWCOUNT ON|OFF** – In the past, this statement has been used to set the maximum row count for the result set. However this feature is degraded in future versions of SQL and should not be used. Use “TOP *n*” in the SQL statement to achieve the same result.

## Code formatting

Unlike some coding languages, such as Visual Basic, SQL is keyword based and does not terminate a statement with a carriage return. For this reason, code formatting is essential to readability of the code. some general guidelines to help standardize the formatting of statements for readability.

### Indention

The following indention guidelines should be followed:

* All indentions should be indented using tab characters in preference to spaces.
* Statements should be formatted consistently regardless of object type.
  + Each clause of a statement should start on a separate line and the intention of the clause’s primary keyword should be equal for a statement.
  + Comma separated list should be one column per line, indented from the parent clause and preceded with a comma
  + Table joins in the FROM clause should be on a separate line and indented from the FROM clause. The join predicate or ON clause should be on a separate line from the join and indented.
  + Subqueries should follow the same formatting rules as a single statement, but be aligned under the SELECT keyword.
  + Each search argument should be on a separate line and indented from the WHERE clause.

See the following example:

SELECT<alias>.<Column\_Name>

,<alias>.<Column\_name>

FROM<Scehma>.<Table\_Name>t1

[LEFT|RIGHT|FULL|INNER|CROSS] JOIN<scehma>.<Table\_name>t2

ON<Join\_Predicate>

[LEFT|RIGHT|FULL|INNER|CROSS] JOIN<scehma>.<Table\_name>t3

ON<Join\_Predicate>

WHERE<Filter>

AND<Filter>

EXISTS( SELECT 1

FROM<Table\_Name>

WHERE<Filter> )

GROUPBY<Column\_List>

HAVING<Predicate>

ORDERBY<Column\_List>

### Capitalization

While SQL is not case-sensitive, capitalization is further used to maintain readability. The following guidelines should be used in all code types.

* SQL keywords used to define clauses should be upper case.
* Capitalization of referenced database objects should match the object. This includes; schema, tables, columns, and database code objects.
* Table aliases should be in lower case to set them apart from the schema names

### Parameterization

To aid in reuse of stored procedures, the parameter names should convey their meaning without having to refer to documentation. The following guidelines are recommended for all parameter declarations.

* Parameter names should match the column they reference in the database. Where a column name is in high use, such as identity columns, the name should be qualified to so that the developer knows how the parameter is used.
* All parameters are assumed to be input parameters unless otherwise specified. Output parameters should be prefixed with “o” and declared last to set them apart from other parameters.
* Parameter data types should match the underlying column that they reference. Data type mismatches run the risk of a SARG not being deterministic which could cause performance issues.
* All declared parameters should be used in the code.

# Development best practices

A database manages the information about a business and database code objects are used to maintain that information. Development of database code objects should have the following design goals

* *Design code for reuse* Code objects can make up a large part of an application’s code base; and since multiple applications usually consume the same database, the number of code objects can easily grow in to the multiple thousands.
* *Limit code objects to CRUD operations* – Code objects should limited to CRUD (Create Retrieve, Update and Delete) operations and business logic and processing should be done at the application level. This minimizes the need to change code objects as the business changes and eliminates.
* *Plan for performance*. – Multi user applications that consume shared database resources, and a single code object performing badly can affect all of the systems. Therefore, database code should be designed to minimize the amount and of resources and the time it holds a resource.

## Query processing

A SQL statement is made up of different clauses. The following list guidelines for each clause.

### Record set (SELECT clause)

The SELECT clause lists the columns returned by a query. The following guidelines should be used when creating a select list

* Always list all columns needed in a select clause. SELECT \* is not allowed.
* All columns should be qualified with the table name or table alias. Table Alias is preferred.
* DISTINCT should be used sparingly. The GROUP BY clause is preferred to distinct.
* Subqueries and scalar functions that access tables are not allowed in the SELECT clause. These should be part of the FROM clause
* It is important to not that TOP is processed after the FROM clause so the execution plan would contain the same number of records. However, when returning a large record set, TOP can still help performance during paging by minimizing network traffic

### Joining Tables (FROM clause)

The FROM clause defines the tables and relationships used to retrieve a record set. The following guidelines should be used when creating the FROM clause.

* All tables should be qualified with the schema name and aliased to help readability.
* All joins should use the ANSI standard syntax. Comma separated table list, and “\*=” or “=\*’ should not be used.
* Join predicates should be limited to referential integrity except when a filter is needed to further qualify the child in an outer join. Consider the following example:

FROM dbo.SITE s

LEFTJOIN dbo.UserSal sal

ON s.Site\_Id = sal.Site\_id

AND sal.Client\_Id =@Client\_Id-- Defines the R/I

WHERE s.Closed = 0 -- Does not define R/I

The “Client\_Id = @Client\_id” qualifies the left outer join and therefore is part of the join predicate, but the “s.Closed = 0” does not define the referential integrity of the join so it is not part of the FROM clause.

* Avoid using OR in a join predicate since MsSQL can only use an index on one side of an OR predicate, which would degrade performance. Use a sub-query instead.
* Avoid using Table hints. In some cases using a NOLOCK hint is acceptable, if all other alternatives are exhausted. Other Join hints are not acceptable in the FROM clause and Index hints are never acceptable.
* Do not use subqueries in the FROM clause to check if a record exists. Use EXISTS predicate in the WHERE clause.

### Filtering Data (WHERE Clause)

The WHERE clause is used to filter the record set generated in the FROM clause. The following guidelines should be used when defining a WHERE clause.

* All Search ARGumentS (SARGs) should be defined in the WHERE clause instead of the JOIN clause.
* Prefer EXISTS over IN to check for the existence of a record.

### Grouping Data (GROUP BY Clause)

The GROUP BY clause is used to aggregate or summarize the data in a distinct record set. The following guidelines should be used when defining a GROUP BY clause.

* The WHERE clause should be used to GROUP BY is preferred to SELECT DISTINCT.

### Ordering data (ORDER BY Clause)

The ORDER BY clause is used to order the final record set of a query. The following guidelines should be used when designing an ORDER BY clause.

* Do not rely on the database to order data. If a specific order is desired in the data, provide an ORDER BY clause.

## Procedural logic

TSQL uses common production logic, such as WHILE loops, IF statements and CURSORS to control the flow of stored procedures, triggers, and functions. These commands follow the same general rules as other programming languages. The following guidelines should be used during design of code objects that use procedure logic.

### Cursors

Avoid cursors at all cost. Due to the overhead in creating and managing a cursor in TSQL there are VERY inefficient. All of the following options must be eliminated before a cursor is considered acceptable.

* Set Based Solution. TSQL operates best of sets of data so a Set based solution is generally the best alternative to a Cursor.
* WHILE Loop – Generally, a While loop is faster and uses less resources than a cursor. While loops are a good option if an incremental value is available in the data set that can act as a counter.
* Push the loop to the Application. TSQL performs best on sets of data. Looping though sets of data generally results in longer than desired blocking times and excessive I/O. If set based and WHILE loops are not possible then the loop should probably be managed by the application instead of MsSQL.

### WHILE Loops

SQL is a set based language and all loops through data should be treated with suspicion, but there are times when a WHILE loop is needed to process a set of data. In these cases the following criteria should be followed when crating a WHILE loop:

* Commands Executed in the loop should always be encapsulated in “BEGIN” and “END”. TSQL loops with a single command do not require the Begin / End statement but for readability it is preferred that they are always used
* Avoid using SELECT statements in the WHILE condition. For performance, it is preferred that values to be used in the select statement be put into variables before comparing the condition. For example:

-- Good: Max(Col2) is only evaluated once

DECLARE @I INT

,@Counter INT

SELECT

@I =Max(Col2)

,@Counter = 1

FROMTable

Groupby Col1

WHILE ( @Counter != @i )

-- BAD: Max(Col2) is evaluatated for each loop

SET @Counter = 1

WHILE ( @Counter !=(SELECTMax(Col2)FROMTableGroupby Col1))

...

### IF statements

If statements can be used in SQL code objects but care should be taken to ensure that both the true and false path evaluate to the same query plan. For example:

IF ( @X = 1 )

BEGIN

SELECT Col1 from Table1

END

ELSE

BEGIN

SELECT Col1 FROM Table2

END

In the example, the true path executes a statement against Table1 while the false path executes a statement against Table2. In this case the query processor will use a different plan for each statement which could be a performance issue. When this is the case, the IF statement should be executed at the application layer and not the MsSQL layer.

### GOTO statements

Prior to MsSQL 2005, GOTO was commonly used in TSQL code objects to break out of code when there was an error. However, 2005 introduced the TRY / CATCH block which eliminated any valid reason to use a GOTO statement.

## Subqueries, Temporary tables, CTE’s, and Unions

### Subqueries

Subqueries are either self-contained or correlated based on there dependency with the outer query. Self-contained queries are generally used in FROM clause of a statement and can be run independently of the outer statement. Correlated subqueries have references to the outer query and are generally used in the WHERE clause to produce some condition.

#### Self containedsubqueries

In general, self-contained subqueries should be viewed with suspicion. These types of queries can easily be replaced with left outer joins or common table expressions introduced in MsSQL 2005. Self-contained subqueries should only be used if it can be proven that they offer a performance enhancement over the alternatives. Even then, the schema should be questioned to find an alternative.

#### Correlated queries

Correlated sub queries usually combine data from the sub query and the outer query to fine a specific condition and commonly used for “tiebreakers” or checking for an existence of a record.

“Tiebreakers” is a data access pattern where you need to specific record in a group of records. Say you need to find the most recent budget for a client. In this case you could use the following example:

SELECT c.CLIENT\_ID, b.BUDGET\_DATE

FROM Clients c

INNERJOIN Budgets b ON c.Client\_id = b.Client\_Id

WHERE c.CLIENT\_ID = @Client\_Id

b.BUDGET\_DATE =( SELECTMAX(Budget\_Date)

FROM Budget

WHERE CLIENT\_ID = @Client\_Id

GROUPBY CLIENT\_ID, BUDGET\_DATE )

ORDERBY c.CLIENT\_ID, b.BUDGET\_DATE

##### IN and EXISTS

Another use of correlated subqueries is to determine if a condition exists in a database. This is generally done using either IN or EXISTS predicate with a subquery. Use the following guidelines for designing subqueries using IN and EXISTS:

* In almost all cases, and IN predicate that uses a subquery can be converted to a left outer join, which is preferred. .
* Use EXISTS when you need to determine the existence of a record.
* When using EXISTS, the SELECT clause in not evaluated so use “SELECT 1” instead of “SELECT <Column\_List>” in the subquery.

### Temporary Tables and Table Variables

Temporary tables come in two forms, Table objects that reside in TempDb and Table Variables that reside in memory. Both of them have there place in MsSQL and this is generally dependent of the size of the temporary table. Use the following guidelines for creating both

* Use temporary tables and table variables sparingly. Prefer common table expressions (CTE’s) and Subqueries to temporary tables and table variables.
* Prefer table variables to temporary tables. Temporary tables are less efficient since they reside in TempDB and require physical I/O to generate, populate and remove. Table variables are created in memory and therefore do not have the I/O overhead.
* Use temporary tables when large data sets are involved. Temporary tables are useful when you are dealing with a large set of data that must be manipulated
* Temporary tables should always be defined before they are populated .

### Common Table Expressions (CTE)

Common table expressions, or CTE’s, where part of the ANSI SQL: 1999 standard and implemented in SQL 2005. CTE’s are similar to temporary and derived tables but have several advantages over there predecessors. The biggest advantage over derived tables is to write recursive statements. The following guidelines will help you choose CTE’s over temporary tables and subqueries.

* Use CTE’s when the same subquery is used multiple times in a statement. CTE’s can be defined once and used multiple times by there outer query. In this case, CTE’s are both easier to read and better performing.
* Use CTE’s where there are many subqueries in the same statement. While this may not have a performance gain, it does make it easier to read by breaking down each subquery into logical statement blocks.
* The biggest advantage of CTE’s is that they allow for writing recursive queries. This is made possible because CTE’s can reference themselves.

### Unions

Unions combine the results of two record sets into a single record set. There are two types of unions: UNION and UNION ALL. The UNION Statement generates a single distinct record set, while UNION ALL creates a single records set containing duplicates. For obvious reasons, UNION ALL is preferred over UNION unless a distinct record set is absolutely needed.

* Unions should always be treated with suspicion as they usually point to flaws in the data model.

## Data Modification statements

In terms of business knowledge, data is the lowest common denominator. If data is incorrect, then business knowledge is incorrect. For this reason we need to take data modifications very seriously by ensuring the modifications are made accurately and as expected. This is made more challenging by the disconnected record sets used in today’s internet architecture. For this reason, there are several guidelines that should be used when developing data modification code.

* When a disconnected record set is used to update data in a database; check that the data has not changed since it was retrieved before updating. In the case of data that is frequently updated, a Row Version column should be used to validate serialization.
* Data modifications (Insert and Update) should be wrapped in a TRY / CATCH block or an explicit transaction to ensure the modification was successful. Never rely on a record set from the database to validate an insert or update.
* When multiple data modification statements are being executed on related tables; an explicit transaction should be used to ensure all changes are complete before committing. The transaction should be started by the database code and not the application.
* Only update records in the source that have changes in the disconnected record set. Most modern programming languages have a flag to determine if data has been updated since it was retrieved. This flag should be used to determine what e records should be updated in the database.

**5.0 File Organization/Versioning**

5.1 Team Foundation Server[Any other source safe tools]

Scripts Files for the database objects (Stored procedures, Function, Triggers, etc.) should always be stored in Team Foundation Server/any other source safe tools. check in/out comment is mandatory.

5.2 Project Documentation

All Project level documentation will reside in a central location like shared server, SharePoint etc. Project documentation includes items such as Business Requirements, Detailed Designs, etc.

**6.0 Summary**

**Structure**

* Each table must have a primary key
  + In most cases it should be an IDENTITY column named ID
* Normalize data to third normal form
  + Do not compromise on performance to reach third normal form. Sometimes, a little de-normalization results in better performance.
* Do not use TEXT as a data type; use the maximum allowed characters of VARCHAR instead
* In VARCHAR data columns, do not default to NULL; use an empty string instead
* Columns with default values should not allow NULLs
* As much as possible, create stored procedures on the same database as the main tables they will be accessing

**Formatting**

* Use upper case for all SQL keywords
  + SELECT, INSERT, UPDATE, WHERE, AND, OR, LIKE, etc.
* Indent code to improve readability
* Comment code blocks that are not easily understandable
  + Use single-line comment markers(–)
  + Reserve multi-line comments (/\*.. ..\*/) for blocking out sections of code
* Use single quote characters to delimit strings.
  + Nest single quotes to express a single quote or apostrophe within a string
    - For example,SET @sExample = ‘SQL”s Authority’
* Use parentheses to increase readability
  + WHERE (color=’red’ AND (size = 1 OR size = 2))
* Use BEGIN..END blocks only when multiple statements are present within a conditional code segment.
* Use one blank line to separate code sections.
* Use spaces so that expressions read like sentences.
  + fillfactor = 25, not fillfactor=25
* Format JOIN operations using indents
  + Also, use ANSI Joins instead of old style joins4
* Place SET statements before any executing code in the procedure.

**Coding**

* Do not use SELECT \*,Use Column Lists
* Return multiple result sets from one stored procedure to avoid trips from the application server to SQL server
* Avoid unnecessary use of temporary tables
  + Use ‘Derived tables’ or CTE (Common Table Expressions) wherever possible, as they perform better6
* Avoid using <> as a comparison operator
  + Use ID IN(1,3,4,5) instead of ID <> 2
* Use SET NOCOUNT ON at the beginning of stored procedures7
* Do not use cursors or application loops to do inserts8
  + Instead, use INSERT INTO
* Fully qualify tables and column names in JOINs
* Fully qualify all stored procedure and table references in stored procedures.
* Do not define default values for parameters.
  + If a default is needed, the front end will supply the value.
* Do not use the RECOMPILE option for stored procedures.
* Place all DECLARE statements before any other code in the procedure.
* Do not use column numbers in the ORDER BY clause.
* Do not use GOTO.
* Check the global variable @@ERROR immediately after executing a data manipulation statement (like INSERT/UPDATE/DELETE), so that you can rollback the transaction if an error occurs
  + Or use TRY/CATCH
* Do basic validations in the front-end itself during data entry
* Off-load 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
* Always use a column list in your INSERT statements.
  + This helps avoid problems when the table structure changes (like adding or dropping a column).
* Minimize the use of NULLs, as they often confuse front-end applications, unless the applications are coded intelligently to eliminate NULLs or convert the NULLs into some other form.
  + Any expression that deals with NULL results in a NULL output.
  + The ISNULL and COALESCE functions are helpful in dealing with NULL values.
* Do not use the identitycol or rowguidcol.
* Avoid the use of cross joins, if possible.
* When executing an UPDATE or DELETE statement, use the primary key in the WHERE condition, if possible. This reduces error possibilities.
* Avoid using TEXT or NTEXT datatypes for storing large textual data.9
  + Use the maximum allowed characters of VARCHAR instead
* Avoid dynamic SQLstatementsas much as possible.10
* Access tables in the same order in your stored procedures and triggers consistently.11
* Do not call functions repeatedlywithin your stored procedures, triggers, functions and batches.12
* Default constraints must be defined at the column level.
* Avoid wild-card characters at the beginning of a word while searching using the LIKE keyword, as these results in an index scan, which defeats the purpose of an index.
* Define all constraints, other than defaults, at the table level.
* When a result set is not needed, use syntax that does not return a result set.13
* Avoid rules, database level defaults that must be bound or user-defined data types. While these are legitimate database constructs, opt for constraints and column defaults to hold the database consistent for development and conversion coding.
* Constraints that apply to more than one column must be defined at the table level.
* Use the CHAR data type for a column only when the column is non-nullable.14
* Do not use white space in identifiers.
* The RETURN statement is meant for returning the execution status only, but not data.

***7.0 RulesFor generating Scripts***

***Stored Procedure***

IF EXISTS(SELECT 1 FROM sys.Objects WHERE name=N'USP\_Add\_UserEffectiveDate\_Case' AND type=N'P')

DROP PROC dbo.USP\_Add\_UserEffectiveDate\_Case

GO

SET ANSI\_NULLS ON

GO

SET QUOTED\_IDENTIFIER ON

GO

-- =============================================

-- Author: <Mahesh KM>

-- Create date: <July-03-2013>

-- Description: Insert case if program are different , and create a record in user\_effectiveDate table with new connectionid or passed connid

-- =============================================

CREATE PROCEDURE dbo.USP\_Add\_UserEffectiveDate\_Case

(

@ConnId AS Int,

@ConnectionId AS Int,

@GroupId as Int

)

AS

BEGIN

SET NOCOUNT ON;

DECLARE @User\_Id as int ,

@RunSql as nvarchar(4000),

@ParamDef as nvarchar(1000),

@strDBName as Varchar(10),

@CAccountKey as Varchar(10),

@ConnectionIdNew as int,

@grp\_ProgramId as int,

@Rel\_ProgramId as int

***2) Function***

IF EXISTS (SELECT \* FROM sys.objects WHERE object\_id = OBJECT\_ID(N'[dbo].[udf\_FamilyConnId]') AND type in (N'FN', N'IF', N'TF', N'FS', N'FT'))

DROP FUNCTION [dbo].[udf\_FamilyConnId]

Go

SET ANSI\_NULLS ON

GO

SET QUOTED\_IDENTIFIER OFF

GO

-- =============================================

-- Author: <VINCE>

-- Create date: <July-03-2013>

-- Description: Get Family ConnIds

-- =============================================

Create function [dbo].[udf\_FamilyConnId] (

@ConnId varchar(20))***s***

***7.0 Check List***

|  |  |  |
| --- | --- | --- |
|  | ***Objects*** |  |
| ***1*** | Applied Naming Rules |  |
|  |  |  |
|  | ***Tables*** |  |
| ***2*** | Primary Key with clustered index |  |
| ***3*** | Data Integrity |  |
|  |  |  |
|  | ***Stored Procedure/Query*** |  |
| ***5*** | *Use upper case for all SQL keywords* |  |
| ***6*** | *SET NOCOUNT ON at the beginning of stored procedures* |  |
| ***7*** | *Code object header* |  |
| ***8*** | *Indent code* |  |
| ***9*** | *BEGIN..END Blocks* |  |
| ***10*** | *Try - Catch* |  |
| ***11*** | *Return Status* |  |
| ***12*** | *All tables qualified with the schema name and aliase* |  |
| ***13*** | *All joins using the ANSI standard syntax* |  |
| ***14*** | *Place all DECLARE statements before any other code in the procedure* |  |
| ***15*** | *Using Column Names instead of SELECT \** |  |
| ***16*** | *Using while loops instead of cursors* |  |
| ***17*** | *Removed the cross joins* |  |
| ***18*** | *Avoid using TEXT or NTEXT data types for storing large textual data.* |  |
| ***19*** | *Avoid dynamic SQL statements as much as possible* |  |
| ***20*** | *Removed RECOMPILE keyword from stored procedures* |  |
| ***21*** | *Removed GOTO keyword from stored procedures* |  |
| ***22*** | *using column names in the ORDER BY clause instead of column number* |  |
| ***23*** | *using column list in the INSERT statements* |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |