MS SQL 2000

Coding standards

&

Best Practices

Version 1.0

Table of Contents

1.0 Introduction 3

2.0 Scope 3

3.0 Development Process 3

3.1 DB Object Naming Conventions (Tables, Procedure, Triggers, Views and Functions) 3

3.1.1 Table names 4
3.1.2 Column names 4
3.1.3 Procedure names 4
3.1.3.1. Procedure variable names 4
3.1.4 Trigger Names 5
3.1.5 Index names 5
3.1.6 Primary and Foreign Key Names 5
3.1.7 Constraint Names 6
3.1.8 View Names 6
3.1.9 Source File Names 6

3.2 General Tip for writing SQL Codes 6

3.2.1 Object creation scripts 6
3.2.1.1. One File Per Object 6
3.2.1.2. Use Database 6
3.2.1.3. Dropping Objects 7
3.2.1.4. Table Creation, Indexes, Triggers 7
3.2.1.5. PK, FK, Check Constraints and Defaults 7
3.2.1.6. Grant Statements 8
3.2.2 Text file format 8
3.2.2.1. Tab Characters 8
3.2.2.2. Indent Spacing 8
3.2.2.3. Format of SQL code 8
3.2.2.4. Specifying Column List 9
3.2.2.5. Headers for script files 9
3.2.2.6. Comments 10
3.2.3 ANSI Compliance 10
3.2.4 Error handling 10

4.0 Guidelines for SQL program units 11

4.1 Basic guidelines 11

4.2 Transactions 14

5.0 Security 14

5.1 Integrated Security 15

5.2 All Database Access through Stored Procedures 15

APPENDIX A 16

APPENDIX B 17

1.0 Introduction

This document provides guidelines and examples of good practices and standards that should be followed when coding SQL in MS SQL 2000 and ASE (Database from Sybase Inc.). These guidelines cover topics such as code layout, naming conventions, variable declarations, and tips for writing efficient maintainable programs.

2.0 Scope

1. General Tips for writing Sql
2. Tips for developing Sql & T-SQL program units
3. Tuning Tips for Sqls and T-SQL in MS SQL 2K and ASE.

3.0 Development Process

3.1 DB Object Naming Conventions (Tables, Procedure, Triggers, Views and Functions)

Even though MS SQL 2000 allows object names to be of 256 characters in length it is recommended to use no more than 30 characters to name DB objects for simplicity, manageability and readability. Database objects should be named clearly and unambiguously. The name should provide as much information as possible about an object and provide an obvious connection to references in project documentation. Choice of names should typically consider a name space much larger than a project—preferably, the enterprise. A name uniquely identifies a database object; its definition should be consistent wherever the object is used.

Naming conventions for specific database objects are described in the following sections. The following guidelines apply to naming *all* database objects:

1. Singular names should be used, implying a single instance of an object.
2. Abbreviations should be avoided unless absolutely necessary due to length restrictions. Abbreviations should remain consistent throughout a project and be documented.
3. If attribute names are going to be used as XML tags through the use of FOR XML within your SQL, you may want to consider using naming conventions that minimize, within reason, the lengths of these names. Depending on the application, there may be a compelling enough reason to keep the xml compact to cause you to consider short names for attributes.
4. Form names by using an initial capital as the delimiter between words, without embedded spaces or underscores. For instance, InvoiceDate.

3.1.1 Table names

Table name should be based on the entity name in the logical model. Table name should clearly shows what kind of data it will stored. For eg. Entity ‘employee’ in logical model will have table ‘Employee’ in physical model. Entity ‘store location’ will correspond to ‘StoreLocation’ in physical model.

3.1.2 Column names

Column names should be based on the attribute names in the logical model. While naming columns consider role of the column in the table e.g. whether it is a primary key, alternate primary key, foreign key or simple data.

E.g. EmployeeId, StoreId, LocationId, EmployeeLastName/LastName, EmployeeAge/Age, QuantityInStock

3.1.3 Procedure names

Name stored procedures using the convention as described above for table, and column names. Do not use SQL Server stored procedure group numbers (review SQL Server BOL for details). Restrict names to alphanumeric characters. Form names by capitalizing the first letter of each word.

Stored procedures should be named starting with a “noun” or “object” and ending with a “verb” or “action”. In this way procedures are grouped together by object when displaying sorted lists from the database.

Always strive to use “verbs” or “actions” in a consistent manner throughout a database. For example, don’t mix “Get” and “Fetch”.

Example Stored Procedure Names: TravelUserGet, TravelUserSearchByName, TravelUserValidate, SystemAuditLogTrim, EmployeeGetStatus.

NOTE:

a. Never prefix your stored procedure names with “sp_”. This is used for naming system stored procedures. System stored procedures reside in ‘master’ database in MS SQL server and in ‘sybsystemprocs’ in Sybase ASE.

b. Never create your stored procedures in system databases viz. master, tempdb, model, or sybsystemprocs databases.

3.1.3.1. Procedure variable names

- Name your variables in procedures in block capital letters.
- Code ‘input’ variables before ‘output’ variables
- All input variables should follow basic and common hierarchy. E.g
    - Input variable ‘@EmployeeId’ should be placed before ‘@EmployeeLastName’ while writing stored procedure.

    - For more clarity suffix ‘Out’ for your output variables. E.g ‘@EmployeeSalaryOut’.

- In case these variables are checked against column names map them to their respective column names. E.g @EmployeeId should map to ‘Employee.dbo.EmployeeId’. Do not use @EmpId or @EId to avoid key strokes.

3.1.4 Trigger Names

The naming convention for triggers is as follows:

Tablename {Ins | Upd | Del} (use this order while creating a trigger) where ‘Ins’ is used for an insert trigger, ‘Upd’ is used for an update trigger and ‘Del’ is used for a delete trigger.

Example Trigger Names: CustomerIns, ProductCodeDel, BusinessUnitUpd

It is possible to have the same trigger used for insert, update and delete actions. If you plan to use the same code for all trigger actions, place this code into a single trigger and apply it to all actions:

Create Trigger (triggername) for insert, update, delete on (mytable)

3.1.5 Index names

Index names are unique within a table so it isn’t necessary to include the table name in the index. When looking at execution plans it is helpful to have a hint about the columns being indexed. The naming convention is as follows:

[IX][type (U if unique C if clustered]_[tablename]_[column name(s)]

Guidelines:

- For single column indexes you may be able to spell out the whole column name.
- For composite indexes, abbreviate as best you can.
- For an index that covers all columns in the table, use the word All.
- Using underscores between the column names help readability.
- Don’t really need to number the indexes.

Examples:

- IXUC_Sales_SalesId (clustered unique)
- IXU_Sales_SalesDate (unique)
- IX_Sales_SalesId_MonId_QtrId_WkId (composite index)
- IX_Sales_MonId_YrId (composite index)
- IXC_Sales_SalesId (clustered not unique)

3.1.6 Primary and Foreign Key Names

The naming convention is as follows:

Primary Key names: PK_[tablename]

Example: PK_Invoice

Foreign Key names: FK[#]_[parent tablename]_[child tablename]

Example: FK1_Customer_Invoice

3.1.7 Constraint Names

The naming convention is as follows:

Check constraint name: CHK_[table]_[column]

Example: CHK_Order_SubmitDate

3.1.8 View Names

The prefix “vw” should be used for all views. Examples:

- vwCustomerInfo
- vwSalesByRegion

3.1.9 Cursor names

Cursors are not an independent objects, they are created at run time either through client side programming (client cursors) or inside stored procedures (database triggers). Name your cursors prefixing with ‘Cur’ and functionality they achieve. Examples:

- CurIncrementTax
- CurCalcFringeTax

3.1.10 Source File Names

Source file names should follow the same naming conventions as for database objects -- no special characters; restrict names to alphanumeric; and capitalize first letter of each word. All source files containing SQL code should end in .SQL, not .SP, .TRG, etc.

3.2 General Tip for writing Object creation SQL script

3.2.1 Object creation scripts

3.2.1.1. One File per Object

All objects have a single file for the drop/create. This helps insure that database objects can be versioned and maintained independently.

3.2.1.2. Use Database

Do not put USE <databasename> statements at the beginning of SQL source files. This will lock the creation into a single database. The command line ISQL has a switch which allows for database redirection.

3.2.1.3. Dropping Objects

Include conditional “drop object” statements before creating an object. See templates for various objects.

Sample table drop:

```
IF OBJECTPROPERTY(object_id('dbo.fldlistversion'), 'IsUserTable') = 1
BEGIN
   PRINT 'Dropping Table fldlistversion...'
   DROP TABLE dbo.fldlistversion
END
GO
```

3.2.1.4. Table Creation, Indexes, Triggers

All create statements should be prefaced with a print statement announcing the creation. The create statement should be followed by key declarations, and binding of defaults or rules within the same table create script.

Table/Index/Constraint Create:

```
PRINT 'Creating Table fldlistversion...'
GO
```

All objects create statements must explicitly specify dbo as the object owner. This prevents objects from being inadvertently created with an owner other than dbo.

3.2.1.5. PK, FK, Check Constraints and Defaults

Primary key constraints should be created during the table definition. Check constraint and column default value definitions should be created via ALTER TABLE commands. There should be one file for FK constraint dropping and one for FK constraint creation for each table.

Constraint creation:

```
IF OBJECTPROPERTY(OBJECT_ID('FK_fldlistversion_fldlist'), 'IsForeignKey')
IS NULL
BEGIN
   PRINT 'Adding Foreign Key FK_fldlistversion_fldlist...'
   ALTER TABLE dbo.fldlistversion
   ADD CONSTRAINT FK_fldlistversion_fldlist FOREIGN KEY
         (fldlistid)
         REFERENCES dbo.fldlist (fldlistid)
END
GO
```

Constraint drop:

```
IF OBJECTPROPERTY(OBJECT_ID('FK_fldlistversion_fldlist'), 'IsForeignKey')
= 1
BEGIN
   PRINT 'Dropping Foreign Key FK_fldlistversion_fldlist...'
   ALTER TABLE dbo.fldlistversion DROP CONSTRAINT
FK_fldlistversion_fldlist
```

```
END
GO
```

3.2.1.6. Grant Statements

Grant statements should be in a separate file from the source code for creating the tables, stored procedures, etc. Keeping the grant statements separate will enable you to re-grant all permissions without recreating every object in your database.

Group like grant statements together. For instance, all table permission granting statements could reside in a single file or all grant statements for a particular process can be grouped together in a single file.

3.2.2 Text file format

3.2.2.1. Tab Characters

There shall be no tab characters in a text source file. Most editors, except Notepad, will allow you to configure your tab key to insert spaces instead of the tab character. You should use 3 spaces. This approach will promote consistency with source files across projects, and will ensure documents look good when printed or when they are brought into ISQL/w.

3.2.2.2. Indent Spacing

Indent blocks of code which are blocked within *BEGIN* and *END* statements:

```
IF EXISTS (SELECT ...
           FROM Table ...)
BEGIN
   statement 1
   statement 2
   IF (...)
   BEGIN
      statement 1
   END
END
```

3.2.2.3. Format of SQL code

Keep all coding within the project consistent. Use the same formatting and indents for all pieces of SQL code source files.

Line up all keywords *SELECT, FROM, WHERE, GROUP BY, HAVING* clauses. Line up all columns in a select clause and indent the AND and OR statements.

```
SELECT T1.Column2
       ,T2.Column1
FROM   Table 1 T1
JOIN   Table 2 T2
```

```
ON      T2.Column1 = T1.Column2
AND     T2.Date = '1/1/1994'
```

Line up *BEGIN* and *END* statements along with *IF* and *WHILE* keywords:

```
IF EXISTS (SELECT *
           FROM #tmptable)
BEGIN
   statement1
   statement2
END
ELSE
BEGIN
   statement1
   statement2
END

WHILE (...)
BEGIN
   statement1
   statement2
END
```

3.2.2.4. Specifying Column List

Never rely on the physical order of the columns in making an insert. Always specify the column's order in the insert clause. This approach makes programs easier to debug and maintain.

```
INSERT Table1 (   Column1,
                  Column2,
                  Column3,
          ....)
SELEC             Column1,
                  Column2,
                  Column3,
          ....
FROM              Table2
WHERE ....
```

3.2.2.5. Headers for script files

All source files shall begin with a header. Put the header before the CREATE...AS statement. This way the comments, including change history, will be stored in sysComments and can be useful for comparing against the latest VISUAL SOURCESAFE'd version.

```
/*******************************************************************
** Copyright information
** Protection level: C1/C2/C3
** File: <objectname>.SQL
** Desc: <description of object>.
** Inputs: List of i/p parameters
**
** Usage:<example useage of object> (example call - preferrably that works)
**
** General Algorithm:
**       1) <step 1 of general algorithm used in processing>.
**       2) <step 2 of general algorithm used in processing>.
**       n) <step n of general algorithm used in processing>.
**
** Dependencies:
**       Tables:
**       Procs:
```

```
**
** *****************************************************************
** **      Change History (order descending by date)
** *****************************************************************
** Date:         Author:     Description:
** ----------    --------    -------------------------------------------
** MM/DD/YYYY    <email>     Initial development.
**
*******************************************************************/
```

3.2.2.6. Comments

Comment complicated sections of code just above the section of code at the same indent level. Do not mix comments within a single statement.

```
/*
** Comment at level one
*/
WHILE (...)
BEGIN
    statement 1
    statement 2
    /*
    ** Comment at level 2
    */
    statement 3
END
```

3.2.3 ANSI Compliance

- Use only ANSI style joins. Do not use the Transact SQL outer join operator (*=).
- Do not use double quotes in stored procedures or sql scripts. This is because the ANSI default of SET QUOTED_IDENTIFIERS is ON and will see anything within double quotes as an identifier.

3.2.4 Error handling

Capturing error on the spot it occurs is mandatory. To capture it is strongly recommended to check status of global variable "@@error" after every DML.

```
E.g.    insert into tablename (col1,
                               col2,
                               col3,
                           …...)
        select                 col1,
                               col2,
                               col3,
                             …..
        from                 tablename, ……….

       If (@@error != 0)
```

```
          Begin
                    Select @msg = "Error: insert failed for 'tablename' with error" +
convert(varchar,@error)
                    GOTO ErrorOccured
          End
```

If you intend to capture this error message from your front end application from where this stored procedures was invoked it is good idea to use 'raiserror' command. With the help of 'raiserror' you can generate custom error code and throw user error message. Global variable '@@error' will be set to the error code your code raised.

For e.g.

```
If(@@error != 0)
Begin
         raiserror 20001, "Could not update table employeee'
end
```

4.0 Guidelines for SQL program units

4.1 Basic guidelines

It is much easier to debug, maintain and tune simple SQL that executes a single process. Try to avoid complex joins and unions if a simpler approach can be adopted that doesn't impact the efficiency of the query. Have your code peer review by other team members after you have got it to compile, prior to the formal technical inspection. It is best to get code peer reviewed earlier to highlight any potential problems or performance hits.

- First rule of any coding is that smaller and simpler codes are easy to manage, debug and maintain. Also they are faster.

- Break complex queries into smaller queries using temp tables

- Any 'select' query without 'where' clause will cause table scan and hard disk hits. Always have 'where' clause to filter data which you are interested.

- Never check existence of a table using 'select * from tablename'. Use system stored procedure 'sp_help tablename'.

- Avoid using calculation in 'where' clause. Calculated where clause do not qualify for 'SARGS' (search arguments) and cause table scan even if useful indexex exists.

- To check existence of data use EXISTS instead ‘select count(*) ……. > 1’.
- Optimizer will always do a table scan if more than 20% of rows are affected. So code your ‘where’ clause accordingly.
- Make sure to check integrity of all indexes your application is using.
- Never specify IS NULL or IS NOT NULL on index columns.
- Never compare NULL to anything else.
- Avoid using NOT in any WHERE condition.
- Avoid the use of HAVING; in general, use WHERE predicates instead.
- Minimize the number of times a table is queried and maximize the number of columns updated with a single SQL statement.
- Remember that if two indexes are of equal ranking in the WHERE clause, Optimizer will use the index of the column specified higher in the WHERE clause.
- If two or more indexes are of equal ranking, you can force a particular index to be used. Verify that full index scanning is not happening by studying ‘plan’ of the query.
- Database (ideally) should never have both the table and its associated index (es) located on the same physical disk. If you have better control of hard disk then place tables and indexes on different disks.
- The only candidates for indexing are columns that are mentioned after WHERE and AND in SQL statements.
- When looking at and using the cost-based optimiser, ensure that your tables are ANALYZED to gather important column statistics that aid in the selection of indexes to create.
- When choosing candidate columns for composite indexes, look at those columns that are used in WHERE and AND together during your application. If they are retrieved in WHERE and AND separately as well as together, two single-column indexes may be better.
- Use a composite index in a query only when the leftmost column in that composite index is mentioned in WHERE or AND.
- Always assess your queries using ‘show plan’.
- To delete all rows from a table use ‘truncate table’.
- Try to avoid dynamic sqls in your program. As they are parsed, compiled and optimized at run time which is heavy on system if tables are huge.
- Always use variables for your where clause instead of hard coding the where clause.
- Do not code correlated sub queries in your applications, as they will adversely impact system performance, consume significant amounts of CPU resources, potentially cause CPU bottlenecks, and disallow application scalability. This means if the number of rows in your table in your correlated sub query increases, you personally writing death certificate of your CPUs. Alternately use inline views (sub queries in the ‘from’ clause of your select statements which have been available from version 7.3), which perform orders of magnitude faster and are much more scalable.

- Beware of mismatch between data types of variables and columns. Mismatch data types is more likely to cause table scan. E.g. numeric(10,2) is not same as numeric(11,3)
- Write separate SQL for separate tasks. Do not try to do multiple things in one SQL batch. This to avoid concurrency overhead.
- DO NOT force the index without thorough analysis of the queries, table structures and the indexes available.
- DO NOT force the join order without understanding the table structures, indexes and number of rows in the tables.
- When checking for varchar and text columns do not put '%' before the character string, i.e description like '%anacin%'. Try to be as sure as possible for the where clause.
- Queries with many OR clause, try rewriting them with UNION operator.
- Do not create indexes on columns with low cardinality columns, e.g. Employee.status, where possible values are 'ACTIVE' or 'INACTIVE'.
- Using views in your queries may cause complex join condition underneath. Try to put 'views' join condition in your 'where' clause.
- Both Sybase and MS SQLs cost based optimizer will consider joining of table based on the size of the tables, statistics, and indexes, available. So it is immaterial which way joins are written unless your force the join order.
- Try to use derived tables in place of temp tables (# tables) to improve performance only if rows retrieved are fairly low. Prefer #tables if intermediate result sets are large and reusable.
- Try to put as much code possible in stored procedures then in embedded sqls in front end programming.
- To optimize cursors DO NOT use them. If your table has a unique index use 'while loop' with 'set rowcount 1' to get optimum performance at locking level.
- Always remember that MS SQL and ASE are designed for batch processing and not row by row processing. One insert which affect 1000 rows will be faster than thousand inserts which affects one row.
- DO NOT hard code usernames/passwords, server names, database names etc. Always store them in configuration file and read them.
- DO NOT put DDL in your DML.
- Be very careful while implementing database triggers. If implemented then document properly.
- Do not use SQL for formatting text. Get your rows and do all formatting at front end client side.
- Never use 'SELECT *' in your procedure. If structure of underlying table is altered it will not be reflected in the output of procedure till the procedure is recompiled.
- DO NOT use 'float' data type for financial calculations. Always use numeric(p,s).
- When calling a stored procedure with parameters, it's a good idea to name the parameter and the value instead of just supplying the values in order.
- Wherever possible use 'SET' to set the variables instead 'select'.
-

4.2 Transactions

- Try to keep transactions as small as possible.
- Code proper ‘rollback’ and ‘commit’.
- Beware of incomplete transactions which are neither ‘rolled back’ nor ‘committed’.
- Deadlock is a programming problem and can be avoided with proper transaction handling among multiple programs/procedures.
- Avoid creating unnecessarily large transactions
- Write code that minimizes the amount of work that would have to be redone in case of failure.
- Minimize the amount of time between BEGIN and COMMIT transaction statements
- Beware of using ‘chained’ transaction against default ‘unchained’ transaction.

5.0 Security

Data is a corporate asset. Security must be designed into applications and databases from the beginning of the design process. Secure databases don’t just happen, and effective security controls rarely happen as an afterthought to a development effort. Security must be made a priority in the design process, even if the design becomes more complex as a result of the security controls. Simplicity of design is not an excuse for lack of security controls.

Toward the objective of ensuring the maximum security possible, the following generalized standards should apply:

- User access to a server and database should be implemented using only the built-in SQL Server security features (standard, mixed, integrated security, GRANTS to users or db groups etc.). Only authorized users may be granted access to a server and its database(s) using SQL Server’s built-in security features. For example, allowing all users to login to the server and then use non-system (application-specific) tables to validate a user should be prohibited.

- Database Developer’s should be aliased to DBO in the databases they are developing to ensure unbroken ownership chains.

Grants on objects should fully comply with the project’s security requirements. The security requirements should explicitly specify what grants are *and* are not allowed on all objects for users and/or database groups. Grants not identified by the specification/design should not be granted.

To implement access and rights either use DB level GROUPS or server wide ROLES. NEVER use both in your system. As over period of time it will become too complex to manage.

5.1 Integrated Security

Use integrated security SQL logins where possible. If there is a need to use a standard login, consider the security implications carefully and have this decision reviewed.

5.2 Database Access

At first no user should be allowed to access data server directly by using any client, viz. winsql, universal sql, etc. If in case any user needs direct access to server care should be taken as mentioned below.

Users should never be granted direct insert, update, select or delete access to the underlying tables.

In the case where direct select access is required to support a reporting function, a view should be created to expose only the minimum data set to the user. The appropriate role may then be granted select access to the view.

To change the data a procedures must be created and execute permission must be granted to users.

In case of group of users needs access use database level groups or server wide roles feature of database security. Due care should be taken not to mix 'groups' and 'roles' privileges. Use any one of them but never both.

APPENDIX A

SQL mistakes to avoid

1. Creating tables with no primary key.
2. Creating tables with no clustered index (typically we have clustered primary keys)
3. Creating tables without properly analyzing data types of the columns.
4. Trying to alter the schema of table once the system is ready.
5. Creating too many indexes on a table.
6. Putting complex logic in triggers.
7. Having combination of triggers, rules, defaults, constraints, etc to enforce consistency.
8. Not using transaction processing to ensure the consistency and integrity of the data.
9. Using transactions but not checking for errors and not issuing rollbacks and flow control statements when the error occurs.
10. Ignoring referential integrity by not enforcing the parent-child (or primary key-foreign key) relationships in the database.
11. Not checking the plan of a query before rolling out.
12. Implementing security as an afterthought in the development process or not at all.
13. Developing a system before the database schema is modeled and then reverse engineering the schema automatically after the system is built. Forward engineer!
14. Having data type mismatches between columns that are in a primary/foreign key relationship.
15. Using row processing logic instead of leveraging the set processing power of the relational database engine.
16. Using labels, such as column names, that don’t relate intelligibly to the contents signified by the label.
17. Allowing unintentional and unnecessary redundant data in the database.
18. Not using adequate headers and comments in stored procedures and SQL scripts.
19. Doing the design doc after writing the code or not doing a design doc at all.
20. Not having a design and or code review.
21. Having no explicit and complete operations plan for installation, backups, recovery, maintenance etc.
22. Using undocumented system procedures in your code
23. Using maintenance commands (update statistics, etc) in your program.
24. Change connection setting for individual connections.
25. Creating db objects in system databases.

APPENDIX B

Points to remember

- MS SQL is case insensitive
- Sybase ASE by default is case sensitive.
- You can find Sybase ASE mainly on UNIX platform where lower case is preferred and coding in lower case is preferred way of programmers.
- Few things discussed in this documents regarding Sybase ASE will be available in upcoming version of Sybase ASE 15.0
- This is a live document and will be under going constant change depending on new versions of server release and feedback from programmers and administrators.