

I. Naming Conventions

The main goal of adopting a naming convention for database objects is so that you and others can easily identify the type and purpose of all objects contained in the database. The information presented here serves as a guide for you to follow when naming your database objects. When reading these rules and guidelines, remember that consistent naming can be the most important rule to follow. Please also keep in mind that following the guidelines as outlined in this document can still produce long and cryptic names, but will limit their numbers and impact. However, ultimately your unique situation will dictate the reasonability of your naming convention. The goal of this particular naming convention is to produce practical, legible, concise, unambiguous and consistent names for your database objects.

This section is a generic DBMS-neutral guide for naming common objects. While most databases contain more types of objects than those discussed here (User Defined Types, Functions, Queries, etc.), the 7 types of objects mentioned here are common among all major database systems.

The following types of database objects are discussed here:

1. Tables
2. Columns (incl. Primary, Foreign and Composite Keys)
3. Indexes
4. Constraints
5. Views
6. Stored Procedures
7. Triggers

ALL DATABASE OBJECTS

- Try to limit the name to 50 characters (shorter is better)
- Avoid using underscores even if the system allows it, except where noted in this document. PascalCase notation achieves the same word separation without them and in fewer characters.
- Use only letters or underscores (try to avoid numbers – and limit the use of underscores to meet standards for Constraints, Special-Purpose Indexes and Triggers or unless implementing a modular naming convention as defined in this document).

SQL Server Standards

Version 1.5

- Use a letter as the first character of the name. (don't start names with underscores or numbers)
- Limit the use of abbreviations (can lead to misinterpretation of names)
- Limit the use of acronyms (some acronyms have more than one meaning e.g. "ASP")
- Make the name readable (they shouldn't sound funny when read aloud).
- Avoid using spaces in names even if the system allows it.

1. TABLES

When naming your database tables, give consideration to other steps in the development process. Keep in mind you will most likely have to utilize the names you give your tables several times as part of other objects, for example, procedures, triggers or views may all contain references to the table name. You want to keep the name as simple and short as possible. Some systems enforce character limits on object names also.

Rule 1a (*Singular Names*) - Table names should be singular, for example, "Customer" instead of "Customers". This rule is applicable because tables are patterns for storing an entity as a record – they are analogous to Classes serving up class instances. And if for no other reason than readability, you avoid errors due to the pluralization of English nouns in the process of database development. For instance, activity becomes activities, ox becomes oxen, person becomes people or persons, alumnus becomes alumni, while data remains data.

Rule 1b (*Prefixes*) – Don't use prefixes unless they are deemed necessary to help you organize your tables into related groups or distinguish them from other unrelated tables. Generally speaking, prefixes will cause you to have to type a lot of unnecessary characters.

Do not give your table names prefixes like "tb" or "TBL_" as these are redundant and wordy. It will be obvious which names are the table names in SQL statements because they will always be preceded by the FROM clause of the SELECT statement. In addition, many RDBMS administrative and/or query tools (such as SQL Server Management Studio) visually separate common database objects in the development environment. Also note that Rule 5a provides a means to distinguish views from tables.

SQL Server Standards

Version 1.5

In some cases, your tables might be sharing a schema/database with other tables that are not related in any way. In this case, it is sometimes a good idea to prefix your table names with some characters that group your tables together. For example, for a healthcare application you might give your tables an "Hc" prefix so that all of the tables for that application would appear in alphabetized lists together. Note that even for the prefix, use PascalCase. This is discussed in Rule 1c. Do not use underscores in your prefixes, which is discussed in more depth in Rule 1d. The last kind of prefix that is acceptable is one that allows you to group logical units of tables. A plausible example could entail a large application (30 to 40+ tables) that handled both Payroll and Benefits data. You could prefix the tables dealing with payroll with a "Pay" or "Pri" prefix and give the tables dealing with benefits data a "Ben" or "Bfts" prefix. The goal of both this prefix and the aforementioned shared schema/database prefix is to allow you to group specific tables together alphabetically in lists and distinguish them from unrelated tables. Lastly, if a prefix is used for this purpose, the shared schema/database prefix is a higher grouping level and comes first in the name, for example, "HcPayClients" not "PayHcClients".

Rule 1c (*Notation*) - For all parts of the table name, including prefixes, use Pascal Case. Using this notation will distinguish your table names from SQL keywords (camelCase). For example, "select CustomerId, CustomerName from MyAppGroupTable where CustomerName = '%S'" shows the notation for the table name distinguishing it from the SQL keywords used in the query. PascalCase also reduces the need for underscores to visually separate words in names.

Rule 1d (*Special Characters*) - For table names, underscores should not be used. The underscore character has a place in other object names but, not for tables. Using PascalCase for your table name allows for the upper-case letter to denote the first letter of a new word or name. Thus there is no need to do so with an underscore character. Do not use numbers in your table names either. This usually points to a poorly-designed data model or irregularly-partitioned tables. Do not use spaces in your table names either. While most database systems can handle names that include spaces, systems such as SQL Server require you to add brackets around the name when referencing it (like [table name] for example) which goes against the rule of keeping things as short and simple as possible.

SQL Server Standards

Version 1.5

Rule 1e (*Abbreviations*) - Avoid using abbreviations if possible. Use "Accounts" instead of "Accts" and "Hours" instead of "Hrs". Not everyone will always agree with you on what your abbreviations stand for - and - this makes it simple to read and understand for both developers and non-developers. This rule can be relaxed in the sake of space constraints for junction table names (See Rule 1f). Avoid using acronyms as well. If exceptions to this rule are deemed necessary, ensure that the same convention is followed by all project members.

Rule 1f (*Junction a.k.a Intersection Tables*) - Junction tables, which handle many to many relationships, should be named by concatenating the names of the tables that have a one to many relationship with the junction table. For example, you might have "Doctors" and "Patients" tables. Since doctors can have many patients and patients can have many doctors (specialists) you need a table to hold the data for those relationships in a junction table. This table should be named DoctorPatient". Since this convention can result in lengthy table names, abbreviations sometimes may be used at your discretion.

2. COLUMNS - (*incl. PRIMARY, FOREIGN, AND COMPOSITE KEYS*)

When naming your columns, keep in mind that they are members of the table, so they do not need the any mention of the table name in the name. When writing a query against the table, you should be prefixing the field name with the table name or an alias anyway. Just like with naming tables, avoid using abbreviations, acronyms or special characters. All column names should use PascalCase to distinguish them from SQL keywords (camelCase).

Rule 2a (*Identity Primary Key Fields*) - For fields that are the primary key for a table and uniquely identify each record in the table, the name should simply be [tableName] + "Id"(e.g.in a Customer table, the primary key field would be "CustomerId". A prefix is added mainly because "Id" is a keyword in SQL Server and we would have to wrap it in brackets when referencing it in queries otherwise. Though CustomerId conveys no more information about the field than Customer.Id and is a far wordier implementation, it is still preferable to having to type brackets around "Id".

Rule 2b (*Foreign Key Fields*) - Foreign key fields should have the exact same name as they do in the parent table where the field is the primary. For example, in the Customers table the primary key field might be "CustomerId". In an Orders table where the customer id is

SQL Server Standards

Version 1.5

kept, it would also be "CustomerId". There is one exception to this rule, which is when you have more than one foreign key field per table referencing the same primary key field in another table. In this situation, it might be helpful to add a descriptor before the field name. An example of this is if you had an Address table. You might have another table with foreign key fields like HomeAddressId, WorkAddressId, MailingAddressId, or ShippingAddressId.

This rule combined with rule 2a makes for much more readable SQL:

... File inner join Directory on File.FileID = Directory.FileID ...

whereas this has a lot of repeating and confusing information:

... File inner join Directory on File.FileId_Pk = Directory.FileId_Fk ...

Rule 2c (*Composite Keys*) - If you have tables with composite keys (more than one field makes up the unique value), it's recommended that a seeded identity column is created to use as the primary key for the table.

Rule 2d (*Prefixes*) - Do not prefix your fields with "fld_" or "col_" as it should be obvious in SQL statements which items are columns (before or after the FROM clause). Do not use a data type prefix for the field either, for example, "IntCustomerId" for a numeric type or "VcName" for a varchar type. These "clog up" our naming and add little value; most integer fields can be easily identified as such and character fields would have to be checked for length in the Object Browser anyway.

Rule 2e (*Data Type-Specific Naming*) - Bit fields should be given affirmative boolean names like "IsDeleted", "HasPermission", or "IsValid" so that the meaning of the data in the field is not ambiguous; negative boolean names are harder to read when checking values in T-SQL because of double-negatives (e.g. "Not IsNotDeleted"). If the field holds date and/or time information, the word "Date" or "Time" should appear somewhere in the field name. It is sometimes appropriate to add the unit of time to the field name also, especially if the field holds data like whole numbers ("3" or "20"). Those fields should be named like "RuntimeHours" or "ScheduledMinutes".

Rule 2f (*Field Name Length*) – Field names should be no longer than 50 characters and all should strive for less lengthy names if possible. You should, however, not sacrifice readability for brevity and avoid using abbreviations unless it is absolutely necessary.

SQL Server Standards

Version 1.5

Rule 2g (*Special Characters*) – Field names should contain only letters and numbers. No special characters, underscores or spaces should be used.

3. INDEXES

Indexes will remain named as the SQL Server default, unless the index created is for a special purpose. All primary key fields and foreign key fields will be indexed and named in the SQL Server default. Any other index will be given a name indicating it's purpose.

Rule 3a (*Naming Convention*) – Indexes will remain named as the SQL Server default, unless the index created is for a special purpose, in which case the naming convention for special-purpose indexes follows this structure:

{U/N}IX_{TableName}{SpecialPurpose}

where "U/N" is for unique or non-unique and "IX_" matches the default prefix that SQL Server assigns indexes.

Rule 3b (*Prefixes and Suffixes*) - Avoid putting any prefix other than that specified in Rule 3a before your special-purpose indexes..

4. CONSTRAINTS

Constraints are at the field/column level so the name of the field the constraint is on should be used in the name. The type of constraint (Check, Referential Integrity a.k.a Foreign Key, Primary Key, or Unique) should be noted also. Constraints are also unique to a particular table and field combination, so you should include the table name also to ensure unique constraint names across your set of database tables.

Rule 4a (*Naming Convention*) - The naming convention syntax for constraints looks like this:

{constraint type}{table name}_{field name}

Examples:

1. PkProducts_Id - primary key constraint on the Id field of the Products table
2. FkOrders_ProductId - foreign key constraint on the ProductId field in the Orders table
3. CkCustomers_AccountRepId - check constraint on the AccountRepId

SQL Server Standards

Version 1.5

field in the Customers table

The reason underscores are used here with Pascal Case notation is so that the table name and field name are clearly separated. Without the underscore, it would become easy to get confused about where the table name stops and the field name starts.

Rule 4b (*Prefixes*) A two letter prefix gets applied to the constraint name depending on the type

Primary Key: Pk

Foreign Key: Fk

Check: Ck

Unique: Un

5. VIEWS

Views follow many of the same rules that apply to naming tables. There are only two differences (Rules 5a and 5b). If your view combines entities with a join condition or where clause, be sure to combine the names of the entities that are joined in the name of your view. This is discussed in more depth in Rule 5b.

Rule 5a (*Prefixes*) - While it is pointless to prefix tables, it can be helpful for views. Prefixing your views with "vw" is a helpful reminder that you're dealing with a view, and not a table. Whatever type of prefix you choose to apply, use at least 2 letters and not just "v" because a prefix should use more than one letter or its meaning can be ambiguous.

Rule 5b (*View Types*) - Some views are simply tabular representations of one or more tables with a filter applied or because of security procedures (users given permissions on views instead of the underlying table(s) in some cases). Some views are used to generate report data with more specific values in the WHERE clause. Naming your views should be different depending on the type or purpose of the view. For simple views that just join one or more tables with no selection criteria, combine the names of the tables joined. For example, joining the "Customer" and "StateAndProvince" table to create a view of Customers and their respective geographical data should be given a name like "vwCustomerStateAndProvince". Views created expressly for a report should have an additional prefix of Report applied to them, e.g. vwReportDivisionSalesFor2008.

SQL Server Standards

Version 1.5

6. STORED PROCEDURES

Unlike a lot of the other database objects discussed here, stored procedures are not logically tied to any table or column. Typically though, stored procedures perform one or more common database activities (Read, Insert, Update, and/or Delete) on a table, or another action of some kind. Since stored procedures always perform some type of operation, it makes sense to use a name that describes the operation they perform. Use a verb to describe the type of operation, followed by the table(s) the operations occur on.

Rule 6a (*Prefixes or Suffixes*) - The way you name your stored procedures depends on how you want to group them within a listing. It is recommended that you have your procedures ordered by the table/business entity they perform a database operation on, and adding the database activity " Get, Save, or Delete" as a suffix, e.g., ("ProductInfoGet" or "OrdersSave") or ("spProductInfoGet" or "spOrdersSave").

If your procedure returns a scalar value, or performs an operation like validation, you should use the verb and noun combination. For example, "ValidateLogin".

The use of the "sp" prefix is acceptable and encouraged. This will help developers identify stored procedures and differentiate them from other non prefixed objects such as tables when viewing a listing of database objects.

Rule 6b (*Grouping Prefixes*) - If you have many stored procedures, you might want to consider using a grouping prefix that can be used to identify which parts of an application the stored procedure is used by. For example, a "PrI" prefix for Payroll related procedures or a "Hr" prefix for Human Resources related procedures can be helpful. This prefix would come before the table/business entity prefix (See Rule 6a). Please note that this should be avoided unless absolutely necessary.

Rule 6c (*Bad Prefixes*) - Do not prefix your stored procedures with something that will cause the system to think it is a system procedure. For example, in SQL Server, if you start a procedure with "sp_", "xp_" or "dt_" it will cause SQL Server to check the master database for this procedure first, causing a performance hit. Spend a little time researching if any of the prefixes you are thinking of using are known by the system and avoid using them if they are.

SQL Server Standards

Version 1.5

7. FUNCTIONS

Functions follow many of the same rules that apply to naming stored procedures. There are only two differences (Rules 5a and 5b) that exist so the user of the function knows they are dealing with a function and not a stored procedure and all of the details that involves (value returned, can be used in a select statement, etc.).

Rule 7a (*Prefixes*) - While it is makes little sense to prefix stored procedures, it is necessary to distinguish functions from stored procedures. To that end, you should prefix your functions with "fn" as a helpful reminder that you're dealing with a function, and not a stored procedure.

Rule 7b (*Function Purpose*) – Functions should be named as a verb, because they will always return a value (e.g. "fnGetOpenDate", "fnParseTableToString", "fnFormatZip", etc.).

Rule 7c (*Function Suffix*) – Scalar functions do not need a suffix, but table-valued functions should indicate that they return a table by using a "Table" suffix.

8. TRIGGERS

Triggers have many things in common with stored procedures. However, triggers are different than stored procedures in two important ways. First, triggers don't exist on their own. They are dependent upon a table. So it is wise to include the name of this table in the trigger name. Second, triggers can only execute when an Insert, Update, or Delete happens on one or more of the records in the table. So it also makes sense to include the type of action that will cause the trigger to execute.

Rule 8a (*Prefixes*) - To distinguish triggers from other database objects, it is helpful to add "tr" as a prefix, e.g. "tr_ProductsIns". As long as you include the table name, the operation that executes the trigger (Ins, Upd, or Del) and the "tr" prefix, it should be obvious to someone working with the database what kind of object it is. As with all conventions you use, pick one and remain consistent.

Rule 8b (*Multiple Operations*) - If a trigger handles more than one operation (both INSERT and UPDATE for example) then include both operation abbreviations in your name. For example, "trProductsInsUpd" or "trProductsUpdDel"

SQL Server Standards

Version 1.5

Rule 8c (*Multiple Triggers*) - Some systems allow multiple triggers per operation per table. In this case, you should make sure the names of these triggers are easy to distinguish between, e.g.

"trUsers_ValidateEmailAddress_Ins" and
"trUsers_MakeActionEntries_Ins".

9. User-Defined Data Types

User-defined data types should be avoided whenever possible. They are an added processing overhead whose functionality could typically be accomplished more efficiently with simple data type variables, table variables, or temporary tables.

Rule 9a (*Prefixes and Suffixes*) - To distinguish a user-defined data type from other database objects, it is helpful to add "ud" as a prefix.

Rule 9b (*Special Characters*) – User-defined data type names should contain only letters, numbers and underscores. No special characters or spaces should be used.

10. Variables

In addition to the general naming standards regarding no special characters, no spaces, and limited use of abbreviations and acronyms, common sense should prevail in naming variables; variable names should be meaningful and natural.

Rule 10a (*Name length limit*) – Variable names should describe its purpose and not exceed 50 characters in length.

Rule 10b (*Prefix*) – All variables must begin with the "@" symbol. Do NOT use "@@" to prefix a variable as this signifies a SQL Server system global variable and will affect performance.

Rule 10c (*Case*) – All variables should be written in camelCase, e.g. "@firstName" or "@city" or "@siteId".

Rule 10d (*Special Characters*) – Variable names should contain only letters and numbers. No special characters or spaces should be used.

11. Using Modular Prefixes

When the scope of the database is expected to be enterprise level or scalability is a great concern consider using a modular object naming convention for primary database objects such as stored procedures

SQL Server Standards

Version 1.5

and views. This methodology involves identifying primary modules of the system and assigning each of those modules a prefix. When these prefixes are applied to the names of database objects this allows us to easily locate module specific procedures and database objects for easier modification and debugging.

Example 1:

We have a modular system that deals with students and teacher data separately. We have defined these modules as such:

Module Name	Prefix
Student	STU
Teacher	TEA

As we implement naming conventions for our new objects processing specific to each module we create an easier way to find and view module specific functionality.

spSTU_Attendance_InserUpdate
spTEA_Credentials_Delete
spSTU_ValidateStudentID
spTEA_Address_InsertUpdate
spSTU_Address_InsertUpdate
vwSTU_AllStudents

Example 2:

Should you find the need for even more granularity when implementing your naming conventions, consider using sub-modules as well.

Module Name	Prefix
Reporting	RPT
Data Collection	DTA

Sub Module Name	Prefix
Student	STU
Teacher	TEA

spDTA_STU_Attendance_InserUpdate
spDTA_TEA_Credentials_Delete
vwRPT_STU_StudentAchievementReport
spRPT_TEA_TeacherQualificationsReport

SQL Server Standards

Version 1.5

spDTA_STU_Address_InsertUpdate
vwDTA_STU_ValidateStudent

II. Coding Conventions

1. SQL Statements (Selects)

Use the more readable ANSI-Standard Join clauses (SQL-92 syntax) instead of the old style joins (SQL-89 syntax). The older syntax (in which the WHERE clause handles both the join condition and filtering data) is being phased out SQL Server 2005 and higher versions. **This “old style” way of joining tables will NOT be supported in future versions of SQL Server.**

The first of the following two queries shows the old style join, while the second one shows the new ANSI join syntax:

```
SELECT
a.AuthorId,
t.Title
FROM Titles t, Authors a, TitleAuthor ta
WHERE
a.AuthorId = ta.AuthorId AND
ta.TitleId = t.TitleId AND
t.Title LIKE '%Computer%'
```

```
SELECT
    a.AuthorId,
    t.Title
FROM dbo.Authors a
INNER JOIN dbo.TitleAuthor ta ON
    a.AuthorId = ta.AuthorId
INNER JOIN dbo.Titles t ON
    ta.TitleId = t.TitleId
WHERE
t.Title LIKE '%Computer%'
```

Do not use the “Select *” convention when gathering results from a permanent table, instead specify the field names and bring back only those fields you need; this optimizes query performance and eliminates the possibility of unexpected results when fields are added to a table.

Prefix all table name with the table owner (in most cases “dbo.”). This results in a performance gain as the optimizer does not have to perform a lookup on execution as well as minimizing ambiguities in your T-SQL.

SQL Server Standards

Version 1.5

Use aliases for your table names in most T-SQL statements; a useful convention is to make the alias out of the first or first two letters of each capitalized table name, e.g. "Site" becomes "s" and "SiteType" becomes "st".

2. SQL Statements (Inserts)

Always use a column list in your INSERT statements. This helps in avoiding problems when the table structure changes (like adding or dropping a column). Here's an example that shows the problem.

Consider the following table:

```
CREATE TABLE EuropeanCountries
(
    CountryID int PRIMARY KEY,
    CountryName varchar(25)
)
```

Here's an INSERT statement without a column list , that works perfectly:

```
INSERT INTO EuropeanCountries
SELECT 1, 'Ireland'
```

Now, let's add a new column to this table:

```
ALTER TABLE EuropeanCountries
ADD EuroSupport bit
```

Now if you run the above INSERT statement. You get the following error from SQL Server:

```
Server: Msg 213, Level 16, State 4, Line 1
Insert Error: Column name or number of supplied values does not
match table definition.
```

This problem can be avoided by writing an INSERT statement with a column list as shown below:

```
INSERT INTO EuropeanCountries
(CountryID, CountryName)
SELECT 1, 'England'
```

3. SQL Statements (Formatting)

SQL statements should be arranged in an easy-to-read manner. When statements are written all to one line or not broken into smaller easy-

SQL Server Standards

Version 1.5

to-read chunks, more complicated statements are very hard to decipher. By doing this and aliasing table names when possible, you will make column additions and maintenance of queries much easier. Refer to the examples below.

Confusing SQL:

```
SELECT dbo.DealUnitInvoice.DealUnitInvoiceID,
dbo.DealUnitInvoice.UnitInventoryID, dbo.UnitInventory.UnitID,
dbo.UnitInventory.StockNumber AS [Stock Number], dbo.UnitType.UnitType
AS [Unit Type], ISNULL(dbo.Make.Description, '') AS Make,
ISNULL(dbo.Model.Description, '') AS Model, DATEPART(YEAR,
dbo.Unit.ProductionYear) AS [Year], dbo.UnitType.UnitTypeID,
dbo.MeterType.Description AS MeterType, dbo.UnitInventory.MeterReading,
dbo.UnitInventory.ECMReading, '$' + LTRIM(CONVERT(nvarchar(18),
CAST(dbo.DealUnitInvoice.Price AS decimal(18, 2)))) AS Price, '$' +
LTRIM(CONVERT(nvarchar(18), CAST(dbo.DealUnitInvoice.Cost AS
decimal(18, 2))))
AS Cost, dbo.DealUnitInvoice.IsTradeIn, ISNULL(dbo.Unit.Vin, '') AS
Vin, ISNULL(dbo.Unit.SerialNumber, '') AS SerialNumber,
dbo.UnitInventory.AvailabilityStatusID,
dbo.UnitInventory.SellingStatusID, dbo.UnitInventory.IsNew,
dbo.UnitInventory.UnitPurchaseOrderID,
dbo.UnitInventory.BaseCost, dbo.DealUnitInvoice.DealPacketInvoiceID
FROM dbo.DealUnitInvoice INNER JOIN
dbo.UnitInventory ON dbo.DealUnitInvoice.UnitInventoryID =
dbo.UnitInventory.UnitInventoryID INNER JOIN
dbo.Unit ON dbo.UnitInventory.UnitID = dbo.Unit.UnitID LEFT OUTER JOIN
dbo.MeterType ON dbo.Unit.MeterTypeID = dbo.MeterType.MeterTypeID LEFT
OUTER JOIN
dbo.UnitType ON dbo.UnitInventory.UnitTypeID = dbo.UnitType.UnitTypeID
AND dbo.UnitType.InActive = 0 LEFT OUTER JOIN
dbo.Make ON dbo.Unit.MakeID = dbo.Make.MakeID AND dbo.Make.Inactive = 0
LEFT OUTER JOIN dbo.Model ON dbo.Unit.ModelID = dbo.Model.ModelID AND
dbo.Model.InActive = 0
```

Now look at the same SQL Statement but organized in an easy to read and more maintainable format:

```
SELECT
dui.DealUnitInvoiceID,
dui.UnitInventoryID,
ui.UnitID,
ui.StockNumber [Stock Number],
ut.UnitType AS [Unit Type],
COALESCE(mk.Description, '') Make,
COALESCE(ml.Description, '') Model,
DATEPART(YEAR,u.ProductionYear) [Year],
ut.UnitTypeID,
mt.Description AS MeterType,
ui.MeterReading,
ui.ECMReading,
```

SQL Server Standards

Version 1.5

```
'$' + LTRIM(CONVERT(nvarchar(18),CONVERT(decimal(18, 2),dui.Price)))
Price,
'$' + LTRIM(CONVERT(nvarchar(18),CONVERT(decimal(18, 2),dui.Cost)))
Cost,
dui.IsTradeIn,
COALESCE(u.Vin, '') Vin,
COALESCE(u.SerialNumber, '') SerialNumber,
ui.AvailabilityStatusID,
ui.SellingStatusID,
ui.IsNew,
ui.UnitPurchaseOrderID,
ui.BaseCost,
dui.DealPacketInvoiceID
FROM
dbo.DealUnitInvoice dui
INNER JOIN dbo.UnitInventory ui
ON dui.UnitInventoryID = ui.UnitInventoryID
INNER JOIN dbo.Unit u ON
ON ui.UnitID = u.UnitID
LEFT JOIN dbo.MeterType mt
ON u.MeterTypeID = mt.MeterTypeID
LEFT JOIN dbo.UnitType ut
ON ui.UnitTypeID = ut.UnitTypeID
AND ut.InActive = 0
LEFT JOIN dbo.Make mk
ON u.MakeID = mk.MakeID
AND mk.Inactive = 0
LEFT JOIN dbo.Model ml
ON u.ModelID = ml.ModelID
AND ml.InActive = 0
```

Note how the tables are aliased and joins are clearly laid out in an organized manner.

4. Setting the NOCOUNT Option

Use SET NOCOUNT ON at the beginning of your SQL batches, stored procedures for report output and triggers in production environments, as this suppresses messages like '(1 row(s) affected)' after executing INSERT, UPDATE, DELETE and SELECT statements. This improves the performance of stored procedures by reducing network traffic.

SET NOCOUNT ON is a procedural level instructions and as such there is no need to include a corresponding SET NOCOUNT OFF command as the last statement in the batch. Refer to the example below.

```
CREATE PROCEDURE dbo.spDoStuff AS
SET NOCOUNT ON
SELECT * FROM MyTable
INSERT INTO MyTable(Col1,Col2,Col3)
SELECT 'One', 'Two', 'Three'
```

SQL Server Standards

Version 1.5

```
DELETE FROM MyTable  
GO
```

5. Code Commenting

Important code blocks within stored procedures and user defined functions should be commented. Brief functionality descriptions should be included where important or complicated processing is taking place.

Stored procedures and functions should include at a minimum a header comment with a brief overview of the batches functionality.

Comment syntax:

```
-- single line comment use 2 dashes (--)  
  
/*  
block comments use ( /* ) to begin  
and ( */ ) to close  
*/
```

Block comments are preferred to single line comments (even for commenting a single line) as cut and paste operations may dislocate the double-dash single line comment, whereas the block comments can never be dislocated.

6. Using Code Headers

Stored procedures, triggers and user-defined functions should have a code header. The header can be created using the comment syntax above. This header should give a brief description of the functionality of the procedure as well as any special execution instructions. Also contained in this header there should be a brief definition of any parameters used. Refer to the example below. You may also include an execution example of the function or procedure in the header as well.

```
CREATE PROCEDURE [dbo].[spGbl_ValidateConcurrency]  
@TableName varchar(255),  
@ID int,  
@LastUpdate datetime,  
@IsValid bit OUTPUT  
AS  
/*****  
This procedure validates the concurrency of a record update by taking  
the LastUpdate date passed in and checking it against the current  
LastUpdate date of the record. If they do NOT match the record is not  
updated because someone has updated the record out from under  
the user.*****/
```

SQL Server Standards

Version 1.5

Parameter Definition:

@TableName = Table to be validated.

@ID = Record ID of the current record to be validated.

@LastUpdate = The Last Update Date passed by app to compare with current date value for the record.

@IsValid = Returns the following back to the calling app.

1 = Record is valid. No concurrency issues.

0 = Record is NOT concurrent.

*****/

7. Unicode vs. Non-Unicode Data Types

Use Unicode data types, like NCHAR, NVARCHAR, or NTEXT, ONLY if your database is going to store not just plain English characters, but a variety of characters used all over the world. Use these data types only when they are absolutely needed as they use twice as much space as non-Unicode data types.

8. CHAR vs. VARCHAR Data Types

Use the CHAR data type for a column only when the column is non-nullable. If a CHAR column is nullable, it is treated as a fixed length column in SQL Server 7.0+. So, a CHAR(100), when NULL, will eat up 100 bytes, resulting in space wastage. So, use VARCHAR(100) in this situation. Of course, variable length columns do have a very little processing overhead over fixed length columns. Carefully choose between CHAR and VARCHAR depending up on the length of the data you are going to store.

9. Very Large Strings

If you don't have to store more than 8KB of text, use CHAR(8000) or VARCHAR(8000) data types. If, however, (and only if) you find that you need to create a field to be sized to accommodate string or binary data > 8KB, then use the VARCHAR(MAX) or VARBINARY(MAX) data types respectively. They are fully integrated with native T-SQL functions and do not require special functions like the old TEXT data types did.

10. Data Access and Transactions

Always access tables in the same order in all your stored procedures and triggers consistently. This helps to avoid deadlocks. Other things to keep in mind to avoid deadlocks are: Keep your transactions as short as possible. Touch as little data as possible during a transaction.

SQL Server Standards

Version 1.5

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. Make your front-end applications deadlock-intelligent, that is, these applications should be able to resubmit the transaction in case the previous transaction fails with error 1205. In your applications, process all the results returned by SQL Server immediately so that the locks on the processed rows are released, hence no blocking.

Always check the global variable @@ERROR immediately after executing a data manipulation statement (like INSERT/UPDATE/DELETE), so that you can rollback the transaction in case of an error (@@ERROR will be greater than 0 in case of an error). This is important, because, by default, SQL Server will not rollback all the previous changes within a transaction if a particular statement fails. Executing SET XACT_ABORT ON can change this behavior. The @@ROWCOUNT variable also plays an important role in determining how many rows were affected by a previous data manipulation (also, retrieval) statement, and based on that you could choose to commit or rollback a particular transaction.

Sample transaction:

```
declare @sql_error int
set @sql_error = 0
begin transaction

update dbo.tbCacLic
set
    sintLicCapacity = source.sintNewLicCapacity,
    chrLastUpdtId = 'myLogin',
    dtmLastUpdt = getDate()
from
    dbo.tbCacLic as target
    inner join #dcfsOldLic as source
    on target.intAplySiteId = source.intAplySiteId

select @sql_error = @@error

if @sql_error = 0
begin
    commit transaction
    drop table #dcfsOldLic
end
else
begin
    rollback transaction
end
```

SQL Server Standards

Version 1.5

12. Error Handling

Although some procedural processes may not require additional error handling, as shown above in the Data Access and Transactions section there may be instances within your stored procedures you wish to handle errors more gracefully. You can use error handling to rollback entire transactions if one piece fails, report a single failed process within a procedure back to the calling application...etc. There are some examples below of proper implementation of error handling within a SQL Server stored procedure.

SQL Server 7.0 – 9.0(2005) – Use this methodology to handle errors in versions previous to SQL 2008 (10.0).

- Tally errors. Rollback if failure occurs anywhere in proc.

```
CREATE PROCEDURE dbo.spDoStuff
@var1 int
AS
SET NOCOUNT ON

--declare/initialize error counter
DECLARE @err int
SELECT @err = 0

BEGIN TRAN

DELETE FROM MyTable
WHERE Coll = @var1
SET @err = @err + @@ERROR

INSERT INTO MyOtherTable(Coll)
SELECT @var1
SET @err = @err + @@ERROR

IF @err = 0
BEGIN
    COMMIT TRAN
    RETURN 0
END
ELSE
BEGIN
    ROLLBACK TRAN
    RETURN @err
END
```

- End processing when error is reached and return error to application.

```
CREATE PROCEDURE dbo.spDoStuff
@var1 int
AS
SET NOCOUNT ON

--declare error counter
```

SQL Server Standards

Version 1.5

```
DECLARE @err int

BEGIN TRAN

DELETE FROM MyTable
WHERE Coll = @var1
SET @err = @err + @@ERROR
IF @err > 0 GOTO ERROR_HDLR

INSERT INTO MyOtherTable(Coll)
SELECT @var1
SET @err = @err + @@ERROR
IF @err > 0 GOTO ERROR_HDLR

GOTO END_PROC

-- Handle Errors
ERROR_HDLR:
    ROLLBACK TRANSACTION
    RETURN @err

-- Return & Exit
END_PROC:
    COMMIT TRANSACTION
    RETURN 0
```

SQL Server 10.0(2008) and greater – SQL version 10.0 has added Try/Catch functionality and new functions to report error status to enhance error handling. You may implement this methodology when developing in SQL 10.0 or higher. Example(s) below.

```
CREATE PROCEDURE dbo.spDoStuff
@var1 int
AS
SET NOCOUNT ON
BEGIN TRY
    BEGIN TRAN

        DELETE FROM MyTable
        WHERE Coll = @var1

        INSERT INTO MyOtherTable(Coll)
        SELECT @var1

        COMMIT TRANSACTION
    END TRY
GO
BEGIN CATCH
    SELECT
        ERROR_NUMBER() as ErrorNumber,
        ERROR_MESSAGE() as ErrorMessage
    -- Test XACT_STATE for 1 or -1.
    -- XACT_STATE = 0 means there is no transaction and
    -- a commit or rollback operation would generate an error.
```

SQL Server Standards

Version 1.5

```
-- Test whether the transaction is uncommittable.
IF (XACT_STATE()) = -1
BEGIN
    PRINT
        N'The transaction is in an uncommittable state. ' +
        'Rolling back transaction.'
    ROLLBACK TRAN
END

-- Test whether the transaction is active and valid.
IF (XACT_STATE()) = 1
BEGIN
    PRINT
        N'The transaction is committable. ' +
        'Committing transaction.'
    COMMIT TRAN
END
END CATCH
```

13. Foreign Key and Check Constraints

Unique foreign key constraints should ALWAYS be enforced across alternate keys whenever possible. Not enforcing these constraints will compromise data integrity.

Perform all your referential integrity checks and data validations using constraints (foreign key and check constraints) instead of triggers, as they are faster. Limit the use triggers only for auditing, custom tasks and validations that cannot be performed using constraints.

Constraints save you time as well, as you don't have to write code for these validations, allowing the RDBMS to do all the work for you.

14. Cursor Usage

Try to avoid server side cursors as much as possible. Always stick to a 'set-based approach' instead of a 'procedural approach' for accessing and manipulating data. Cursors can often be avoided by using SELECT statements instead or at worst, temporary tables and/or table variables.

Should the need arise where cursors are the only option, avoid using dynamic and update cursors. Make sure you define your cursors as local, forward only to increase performance and decrease overhead.

Sample Cursor:

```
DECLARE @var1 int
```

```
DECLARE MyCursor CURSOR LOCAL FAST_FORWARD FOR
```

SQL Server Standards

Version 1.5

```
SELECT Col2
FROM MyTable1
WHERE Col1 = 5

OPEN MyCursor
FETCH NEXT FROM MyCursor INTO @var1
WHILE (@@fetch_status <> -1)
BEGIN
    IF (@@fetch_status <> -2)
    BEGIN
        DELETE FROM MyTable2
        WHERE Col1 = @var1
    END
    FETCH NEXT FROM MyCursor INTO @var1
END
CLOSE MyCursor
DEALLOCATE MyCursor
```

15. Temporary Tables and Table Variables

Use temporary tables (e.g. #NslClaims) only when absolutely necessary. When temporary storage is needed within a T-SQL statement or procedure, it's recommended that you use local table variables (e.g. @NslClaims) instead if the amount of data stored is relatively small. This eliminates unnecessary locks on system tables and reduces the number of recompiles on stored procedures. This also increases performance as table variables are created in RAM, which is significantly faster than physical disk. You can increase performance on temporary tables and table variables in some instances by applying indexes. However, SQL 2005 and greater handles indexes on temp tables and variables much better than SQL 2000. Also keep in mind that when using indexes it can decrease performance on inserts and updates so use caution when choosing indexed columns.

16. Parameter Sniffing – Be Warned

With SQL Server version 9.0 and greater the database engine has enhanced parameter sniffing capability according to Microsoft. This functionality should be monitored with caution as we found that it has bugs. Parameter sniffing cannot be turned off...and has been deemed a "feature" of SQL Server. It basically sniffs the values of incoming stored procedure and function parameters and tries to adjust the query optimization plan accordingly. The problem is under many scenarios it adjusts it for the worse and severely affects the performance of the procedure.

There is, however, a work around. See the example below. By using this method you will in effect bypass parameter sniffing. I would

SQL Server Standards

Version 1.5

suggest only using this where you find problems with parameter sniffing.

```
CREATE PROCEDURE dbo.spDoStuff
@var1 int,
@var2 varchar(20)
AS
SET NOCOUNT ON

---Bypass parameter sniffing---
DECLARE @var1a int,
@var2a varchar(20)

SELECT @var1a = @var1,
@var2a = @var2
-----end bypass-----

SELECT *
FROM MyTable
WHERE col1 = @var1a
and col2 = @var2a
```

17. Database Design – Best Practices

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

- Utilize enforced relationships between tables to protect data integrity when possible. Always remember...garbage in....garbage out. The data will only ever be as good as the integrity used to store it.
- Take care to normalize data as much as possible while still achieving peak performance and maintainability. Over-normalization can also be an issue as it can have a severe effect on performance and data maintenance and cause un-needed overhead. Entities and their proposed uses should be studied to formulate a data model that can balance integrity, scalability and performance.
- Make use of auto-numbered surrogate keys on all tables. This can reduce data storage where compound primary and foreign keys are prevalent and simplify relationships between data when normalizing data structures and writing queries.
- **Always** use primary keys on tables! Even if you are enforcing integrity through other means (i.e. business layer, unique table

SQL Server Standards

Version 1.5

index) always put a primary key on a table. Tables with no primary key are a problem waiting to happen.

- Utilize indexes and fill factors correctly. Nothing can speed up data throughput more than correct utilization of clustered indexes, regular indexes and fill factors. This can be tricky business but SQL Server has a small host of tools that can assist. Nothing better than experience when it comes to optimizing queries and insert/update performance. If you are unsure how to proceed with optimizations consult the DBA.
- Database security is always a concern. ISBE has implemented functionality to increase security by obscuring server and database connectivity information using CIAM and IWAS as well as disabling SQL Server functionality that can be used maliciously or used to gain control of a database server. Close attention must be paid when developing front end applications and stored procedures to protect against SQL injection attacks. Make sure to consult the DBA when establishing your database security model.