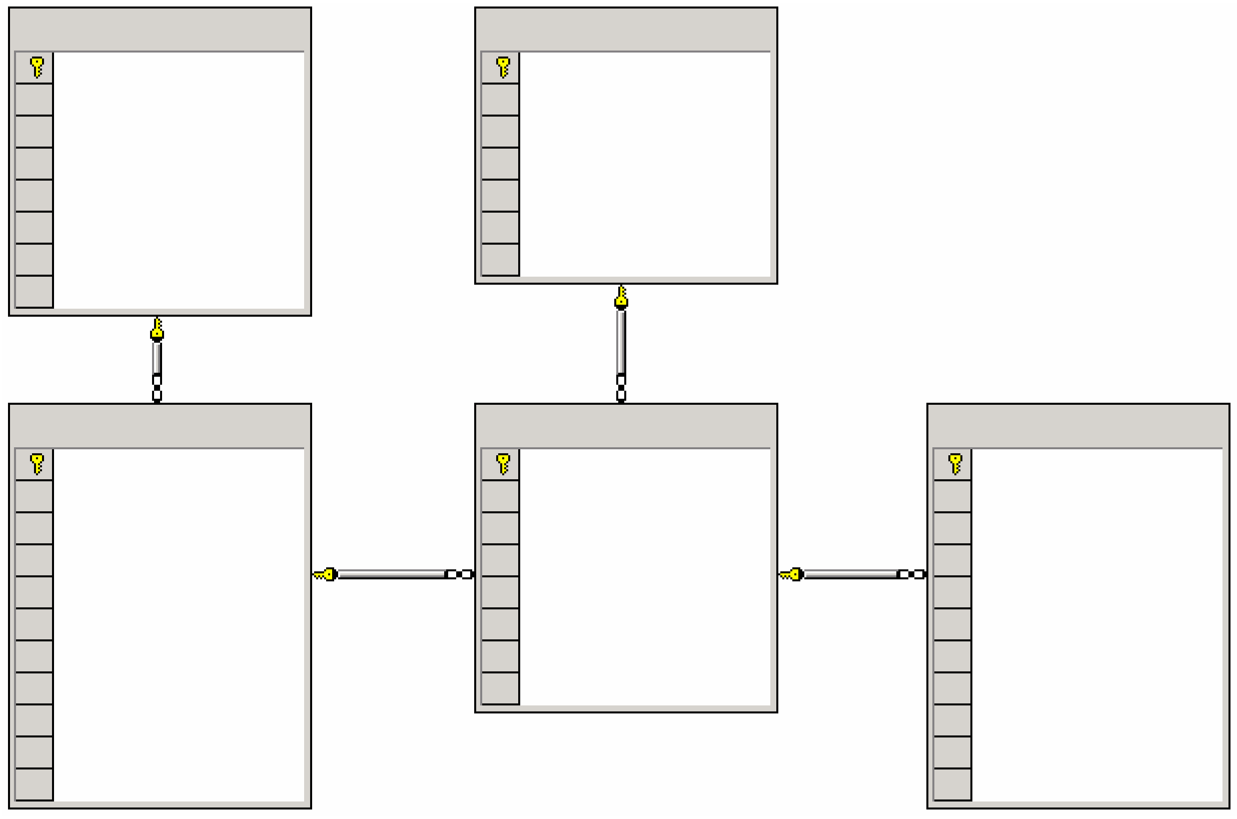


Database Architecture

Fourth Edition

Table of Contents:

Introduction:

The database architecture is the set of specifications, rules, and processes that dictate how data is stored in a database and how data is accessed by components of a system. It includes data types, relationships, and naming conventions. The database architecture describes the organization of all database objects and how they work together. It affects integrity, reliability, scalability, and performance. The database architecture involves anything that defines the nature of the data, the structure of the data, or how the data flows.

This document is intended to be a fairly comprehensive description of a database architecture proposal. The specific database architecture being suggested may not be a perfect fit for every environment. However, even if it does not meet all the needs of a particular situation, it should provide some valuable ideas and important points of consideration.

The database architecture proposed here is the result of much research and practical experience. The advice of many industry experts was gathered from several different sources. That advice was merged with day-to-day experience building the database architecture for a new company from the ground up. The company, a growing data processing services firm (totally separate from Wingenious), is currently using a large custom software package that is based upon a database architecture very much like this one.

This database architecture has served the business mentioned above very well since it was adopted there in 2001. As of late 2005, the company maintains roughly 30 databases on three separate servers. The databases contain roughly 1500 tables with roughly 250 million records. The main database contains about 200 tables with about 50 million records. It's used mainly for OLTP and large batch processing chores, but it also handles some OLAP tasks. DBA duties are greatly simplified and extremely efficient largely due to dynamic routines that are made possible by the consistency of the database architecture.

This database architecture is intended to be generic and applicable to any business. It addresses only the back-end of a system, leaving the front-end choices for others to debate. The various options for presenting data to users are beyond the scope of this document. This document discusses the database itself primarily, but it also touches on getting data to a middle tier of a multi-tier system. The information should be beneficial to a DBA or a software developer, and especially a person whose job includes aspects of both positions.

This document was originally written several years ago and it has not been thoroughly revised since then. Wingenious continues to follow the general principles outlined here, but it does not necessarily follow every specific practice. The standards (such as using prefixes) have evolved over time, but Wingenious remains committed to ensuring consistency within a database.

Background:

This document assumes that the reader is familiar with basic database terminology and usage. The database terms table (file), row (record), column (field), and so forth are not defined here. Most readers interpret the specific word pairs above as synonyms. They are very widely used interchangeably, including in this document.

This document contains some introductory material, but even experienced database developers may find some useful tidbits here and there.

This document was written with SQL Server in mind. It refers exclusively to the objects and tools available in that environment. The vast majority of the topics are applicable to both SQL Server 2000 and SQL Server 7, but a few things are specific to SQL Server 2000. Still, several of the core concepts are also applicable to other relational database systems.

Any lengthy discussion of SQL Server databases eventually involves Transact-SQL (T-SQL). T-SQL is the custom SQL implementation of SQL Server. This document addresses many points in the context of T-SQL. Many elements of the language are mentioned and used to provide a frame of reference for examples. Although familiarity with T-SQL would be very beneficial to understanding, a familiarity with standard SQL would also help.

There are many resources available for information about SQL Server and T-SQL. The Books Online (BOL) provided with SQL Server is an excellent reference. There are dozens of printed books and a few e-books, available from many vendors, covering a variety of SQL Server topics. There are printed newsletters and printed magazines available for subscription. There are e-mail newsletters and e-mail tips available for free. There are several web sites devoted to SQL Server and many major technology news and information web sites have sections devoted to coverage of SQL Server topics. The Microsoft web site for SQL Server at http://www.microsoft.com/sql has a wealth of helpful resources. A quick web search for any SQL Server topic is likely to find many interesting pieces of information.

Several of the SQL Server web sites have discussion forums where questions can be asked and answers can be debated. However, the debate is rarely useful. It's often difficult to distinguish good advice from mere opinions stated as facts. The broader the topic (database architecture is definitely a broad topic), the more divergent the opinions and the more heated the debate. The contentions often dwell on theory and take on a dogmatic tone. This document stresses a very pragmatic approach to database architecture matters.

Data Types:

SQL Server supports a variety of data types. Half of them are better left unused. There are three basic categories of data types: character (string), numeric, and special.

For many applications, string (character) fields contain the majority of the data. SQL Server has several options for string values. The choices are fixed-length (up to 8000), variable-length (up to 8000), and variable-length (up to over two billion). There's also a Unicode format (using two bytes per character) for each of these choices. The Unicode formats provide some compatibility with international operating systems, but they have only half the capacity.

Using variable-length strings (up to 8000) is the preferred choice because of some limitations with the other two choices. This format (SQL Server data types *varchar* and *nvarchar*) provides more than enough capacity for most character storage needs, and it offers the most consistency with the string handling of other languages and environments.

The fixed-length string format (SQL Server data types *char* and *nchar*) is applicable only under very specific circumstances. It should be applied only when every character position will be used in every record (such as with single character values). Any unused character positions are filled with trailing spaces. This can lead to bizarre and inconsistent behavior when comparing and concatenating strings (see below).

The variable-length string format that allows over two billion characters (SQL Server data types *text* and *ntext*) requires special handling in T-SQL. In some situations this string format may be necessary, but it should be fairly rare to need to store massive amounts of raw text in a database. Generally, when such large amounts of text are involved, it becomes necessary to include formatting. It may be better to store formatted documents outside of the database and store only references to the documents in the database. In addition, some front-end environments may not have a native data type capable of handling this string format.

Always keep in mind that T-SQL does something strange with strings. For some operations, it appears to do an automatic trimming of trailing spaces. When checking the length of a string (with the LEN function) or comparing strings, trailing spaces seem to be ignored. However, when concatenating strings, trailing spaces are not ignored. The automatic trimming is an odd behavior that should not be relied upon, especially when there is an explicit way to do the same thing. Further, it's difficult to understand why the odd behavior is implemented inconsistently.

Although numeric fields may not contain the most data for a typical application, they very often contain the most important data. There are two kinds of numbers that an application might use: integer values and decimal values.

SQL Server has four different sizes for integer values. The sizes are 1-byte (*tinyint*), 2-bytes (*smallint*), 4-bytes (*int*), and 8-bytes (*bigint*). When selecting one of the sizes, always choose the larger size if there is any doubt about exceeding the range of possible values for the smaller size. However, be cautious about choosing the 8-byte size. Many front-end environments do not have a native data type for 8-byte integers. A .NET-based front-end does, but using 8-byte integers could prevent other options.

SQL Server has several options for decimal values. The main difference between the options is whether the decimal point is fixed or floating. Unless an application needs astronomically large or microscopically small values, there is no need for floating-point values (SQL Server data types *float* and *real*). The problem with floating-point values is that they are not exact. Some values may not be exactly represented in floating-point format. This can lead to complications with selecting records and rounding the results of calculations. The vast majority of needs for decimal values can be handled using the fixed-point format. SQL Server provides data types specifically for financial purposes (*money* and *smallmoney*), but it's preferable to explicitly define the precision (total number of digits) and scale (digits to the right of the decimal point) for decimal values. This can be done with the SQL Server data type *decimal* (also called *numeric*).

In general, special data types should be avoided. They often complicate front-end development, limit front-end choices, and restrict options for data import, export, and migration. SQL Server data types such as *binary*, *varbinary*, *image*, *timestamp* (or *rowversion*), and *uniqueidentifier* are examples. Many databases do not need the features provided by these data types. However, there are some special data types that are too beneficial to ignore.

The most useful special data types are *datetime* and *smalldatetime*. The two data types differ in their accuracy and storage requirements. The *smalldatetime* data type is usually sufficient. The use of these data types may complicate data import, export, and migration because every system stores dates and times differently. However, these data types store dates and times much more efficiently than string data types, and the layers of interface software between the database and an application do a pretty good job of converting the data to the necessary form.

The special data type *bit* (a Boolean value) is modestly useful under the right circumstances. If a table must contain multiple binary condition attributes, the values can be stored efficiently using *bit* fields. However, if there is only one binary condition attribute, the value can be stored just as efficiently using the *char* data type or the *tinyint* data type. These alternatives to the *bit* data type are more flexible and the *char* data type is best for migration purposes.

The SQL Server *image* data type is another case (like the *text* data type) where it may be better to store objects outside of the database and store only references to the objects in the database. The hassles involved with getting images into the database and retrieving them from the database are rarely offset by the convenience of having them embedded in the database. Further, when large objects are stored in the database they are included in every database backup. If they are stored outside of the database they can be handled more selectively by backup software.

Null Values:

One of the more confusing aspects of programming a database-driven application is the handling of null values. Each development environment (back-end or front-end) seems to have a different definition of a null value and a different way to represent one. In SQL Server, fields of any data type can contain null values. However, the corresponding native data types for many front-end environments do not allow null values. T-SQL itself has two different ways of comparing null values. The method is determined by a connection setting. It's best to use T-SQL syntax that works regardless of the setting (IS NULL, IS NOT NULL, the ISNULL function). The issues with handling null values can be minimized by allowing them in the database only when it's absolutely necessary. Fields that are populated by triggers alone must allow null values and optional foreign key fields must allow null values.

Naming Conventions:

There are many kinds of objects included in the database architecture. Each object has a name. In order to preserve the sanity of DBAs and developers, a naming convention for every object should be established and strictly followed. Naming conventions allow both groups to easily locate objects and immediately understand the nature of objects from the names. Generally, naming conventions involve prefixes and/or suffixes attached to base names.

This database architecture suggests using base names that include one or more meaningful words, with the initial letter of each word capitalized. The base names must be unique within each object type, but they should not be overly long (preferably less than 50 characters). A three-character prefix appears in front of every base name. The prefix uses lower-case letters and it indicates the type of the object. Except for certain field names (discussed below), no object names include a suffix.

Object Type	**Object Prefix**
Table	`tbl`
Stored Procedure	`usp`
User-Defined Function	`udf`
Trigger	`trg`
View	`qry`
Index	`idx`
Key Constraint	`key`
Key Constraint	`keyPK – Primary Key`
Key Constraint	`keyFK – Foreign Key`

This database architecture encourages the use of an additional two-character prefix for certain types of objects. The two characters are in upper-case letters, and they follow the lower-case prefix. The additional prefix can be used to group objects according to business needs. For example, stored procedures may use additional prefixes such as GP (General Purpose), GR (Generated Routine), HR (Human Resources), CS (Customer Service), IT (Information Technology) or anything that helps to identify a logical grouping of these objects.

The use of prefixes on database object names is hotly debated whenever the topic comes up for discussion. Some DBAs argue with an almost religious fervor instead of simply using common sense (this happens with several other database architecture topics as well). Most of the strong objections to using prefixes come from mere personal opinion instead of being based on sound logic. When using prefixes, the additional length of the names has a negligible effect and there are no other technical reasons to avoid the practice. However, there is a sound technical reason to use prefixes on all database object names.

SQL Server maintains records in the sysobjects system table for most kinds of objects. Those records contain names that must be unique. Many objects are directly related to a single table and it only makes sense to name such objects according to the corresponding table name. What other naming scheme could provide the same degree of consistency and predictability? If there were no prefixes to specify the object types then the names would not be unique. Consider an INSERT trigger and a stored procedure to INSERT a record. Both objects pertain to one table only, and both of them deal with the INSERT action. The base name for each object should be the name of the table to which it applies. A prefix on the name of each object would make the name unique. The object name alone would indicate the type of object and the corresponding table. That can be very handy when working with objects by name in T-SQL.

Choosing base names for tables involves an additional consideration. Should the base names be singular or plural? Try to recognize any discernable logic or reason when making such database architecture decisions. In this case, notice that the English language uses several different ways to determine the plural forms of words. Sometimes the plural form depends on how the singular form is spelled. Sometimes the plural form depends on the word itself. Sometimes there is more than one acceptable plural form of a word. Sometimes the plural form is identical to the singular form. Even if this variability were the only reason for avoiding plurals it’s better than having no sound logic or practical reason behind using them. This database architecture suggests using the singular forms of words for table names.

The preceding paragraphs share a common theme. The theme is consistency and predictability. That’s the most important point this document hopes to convey. Above all, any good database architecture should strive for consistency so that everything is predictable. Do not change the naming convention from object to object. Do not change the design rules from table to table. Make sound decisions on such matters and apply the decisions across the entire database. If object names are consistent then DBAs and developers do not have to guess. If the design of every table is predictable then dynamic routines can be written to perform extremely powerful manipulations of tables and their data. Such routines can be very valuable to DBAs.

Fields are conspicuously absent from the list of object types above. This database architecture suggests using a field name prefix that is based on the data type of the field.

Data Type	**Field Prefix**
Bit	`bln`
TinyInt	`byt`
SmallInt	`int`
Int	`lng`
Char/NChar	`str`
VarChar/NVarChar	`str`
Text/NText	`str`
Decimal/Numeric	`dec`
Float/Real	`dec`
Money/SmallMoney	`dec`
DateTime/SmallDateTime	`dtm`
Other	`bin`

These field name prefixes were specifically chosen to be compatible with commonly used BASIC (such as Visual Basic) variable name prefixes. The goal is to provide an application developer with information to choose an appropriate data type for variables. The prefixes can help developers while not compromising database integrity or performance. They do not affect DBA work and they have no negative impact, but they are not universally accepted.

The use of field name prefixes is an even more contentious issue than using prefixes on other database object names. There is some reason to avoid the practice because the data type for a field could change such that it requires a change in the field name. A change in the field name would require a change to any code that references the field. However, if an appropriate amount of requirements discovery is done prior to data modeling and database design, such changes are very rare. Further, it’s very likely that such a change in data type would require a change to the code anyway. Most code is written to declare/define/dimension variables as a particular data type. If a field were to change from a 2-byte integer to a 4-byte integer, without a change in the field name that would force a developer to review the code, it could result in erratic behavior of the code. Such a problem may be very difficult to track down.

The use of field name prefixes can be very helpful in understanding and changing code that references the fields. However, if prefixes are used incorrectly they can be very misleading. Therefore, **use them consistently or do not use them at all.**

Normalization:

Database normalization is a complex topic that deserves much more explanation than what is contained in this document. There are many books that cover the concepts very well. Basically, it boils down to grouping fields appropriately into tables and forming parent/child relationships between the tables. Among the many goals of proper normalization is ensuring data integrity and avoiding data redundancy. The lack of coverage in this document does not mean that the topic is less important. Database normalization is a critical part of good database design.

Relationships:

Database relationships create a hierarchy for the tables. A properly normalized database has a well-organized hierarchy. For each relationship between two tables, one table is the parent and one table is the child. For example, a Customer table may be the parent of an Order table, and in turn, an Order table may be the parent of an OrderDetail table. In this example, the Order table is both a child (of Customer) and a parent (to OrderDetail).

Database relationships are embodied through primary keys in parent tables and foreign keys in child tables. There are many ways this can be implemented and the various options are another source of heated discussion. There is no universally accepted approach, but there is an approach that appears to be the most common. This document describes that approach and provides sound justification for it with technical reasoning. The justification is based on practical considerations rather than blind adherence to theory.

One of the most important keys (pun intended) to this database architecture is the handling of primary keys and foreign keys.

Primary Keys:

A primary key is a field in a table whose value uniquely identifies each record. There are many qualities required of a field in this position of honor. It must not contain any duplicate values and the values must not change. It should also be compact for the best performance. It’s often very difficult to find such characteristics in naturally occurring data. Therefore, this database architecture uses surrogate primary keys.

A surrogate primary key field contains artificially derived values that have no intrinsic meaning. The sole purpose for the values is to uniquely identify each record. The values are derived in a way that prevents duplication, they are never changed, and they are compact (usually integers). The use of surrogate primary keys avoids the inevitable hassles of trying to use natural data for primary keys. It’s also a very important factor in the advantages of this database architecture.

SQL Server provides a convenient field property that supports the use of surrogate primary keys. One field in a table can be assigned the IDENTITY property. When a record is inserted into the table that field is automatically given a value one greater than the previously inserted record. A field with the IDENTITY property does not (normally) allow values to be explicitly provided with an insert or changed with an update.

In this database architecture, the primary key field is the first field in every table. The field is a 4-byte integer (SQL Server data type int). The field uses the IDENTITY property (starting at 1 and incrementing by 1). The field is named with a prefix (lng), the table name (without a prefix), and a suffix (ID). The primary key field for a Customer table would be named lngCustomerID.

Foreign Keys:

A foreign key is a field in a table that refers to parent records in another table. The references are represented by the primary key values of the corresponding parent records. A foreign key field is the same data type as the primary key field of the referenced table. Usually, a foreign key field is named the same as the primary key field of the parent table. This very beneficial convention is called key migration in data modeling terminology.

Child table foreign key references to parent table primary keys embody database relationships.

Relationships (Again):

Some DBAs vehemently argue that primary keys should be composed of natural data. They do not like surrogate primary keys. This opinion seems to be based on a desire to worship theory rather than a rational examination of the issues involved. Natural data is almost never compact and almost never stable (not subject to change). A single field of natural data is often not even unique. The uniqueness requirement is sometimes met by using multiple fields as a composite primary key. However, that worsens the compactness.

The use of natural data for a primary key also has major implications for child tables. In order to ensure uniqueness in their primary keys, all child tables are forced to include the primary key of their parent table along with an additional field (or more) of their own. This approach becomes extremely unwieldy very quickly in a relational database of any depth.

Imagine a database that contains a Customer table, an Order table, an OrderDetail table, and a Product table. All of these tables use natural data primary keys. The Customer table uses three fields (zip code, last name, date/time the account was created) as a composite primary key to try to ensure uniqueness. The Order table uses a Customer foreign key plus date/time the order was placed as a primary key. The OrderDetail table uses an Order foreign key plus a Product foreign key (name) as a primary key. The OrderDetail table has five long fields as a composite primary key, and this database is only three levels deep! Now imagine the query joining these tables in order to produce a packing list. Then picture a database that has a much deeper hierarchy. It should be clear that natural data primary keys are a troublesome database design decision.

Some of those who recommend natural data primary keys make one valid point. Such keys do reliably prevent duplicate data. Surrogate primary keys by themselves do not prevent duplicate data. However, that goal is easily accomplished by adding a unique constraint on the necessary field or fields. For example, adding a unique constraint on zip code, last name, and date/time of the Customer table above would do the same thing while preserving other advantages.

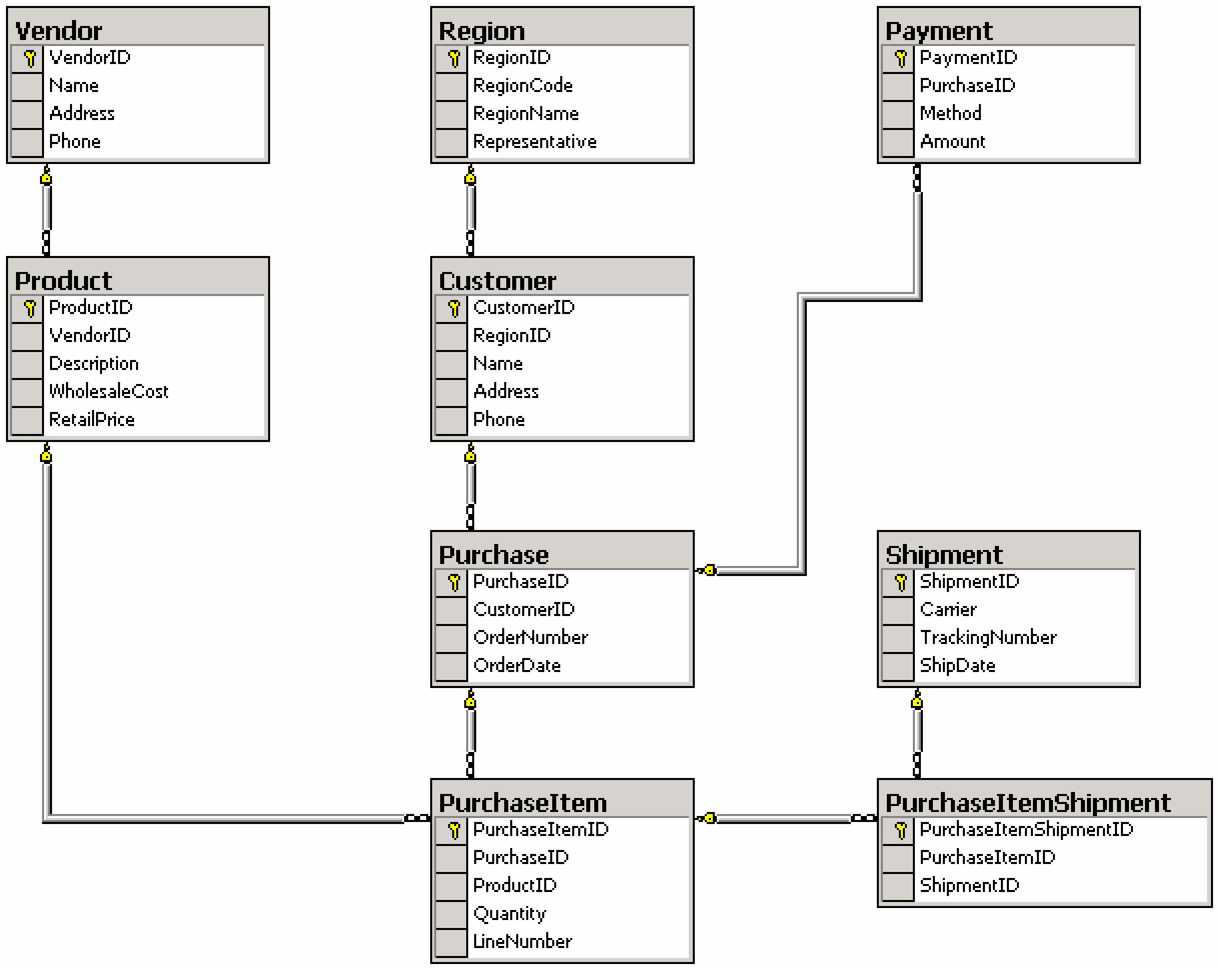
Surrogate primary keys are compact (resulting in efficient indexes), stable (they never change), and consistent. The consistency is a very important point. The database above has a different number of fields in the primary key of every table. The fields are also different data types. A developer would have a difficult time remembering all the details, and leaving a field out of a JOIN could have disastrous consequences. With surrogate primary keys as suggested by this database architecture every JOIN would be simple, obvious, and predictable.

The consistency provided by surrogate primary keys makes it possible to write some extremely powerful dynamic routines for database administration (see General Purpose Routines for some examples). Such routines can be a tremendous time-saver for DBAs. Instead of writing special code for every table generic code can be used with any or all tables.

The rules for the foundation of this database architecture are quite simple. Here they are in a loose order of importance (with 1 being the most important)…

1) Every table has a primary key.
2) The primary key is a single field.
3) The primary key is the first field.
4) The primary key field is named to correspond with the table name.
5) The primary key migrates to child tables as a foreign key with the same characteristics.
6) The primary key field is numeric.
7) The primary key field is a 4-byte integer data type.
8) The primary key field uses the IDENTITY property (starting at 1 and incrementing by 1).

The table definitions below are intended to serve as examples of the rules listed above. They are not intended to be a realistic model for any particular business.



```
CREATE TABLE dbo.Region
   (RegionID                           int IDENTITY(1,1) PRIMARY KEY,
    RegionCode                    char( 2) NOT NULL,
    RegionName                 varchar(40) NOT NULL,
    Representative             varchar(40) NOT NULL)

CREATE TABLE dbo.Customer
   (CustomerID                         int IDENTITY(1,1) PRIMARY KEY,
    RegionID                           int NOT NULL,
    Name                       varchar(80) NOT NULL,
    Address                    varchar(80) NOT NULL,
    Phone                      varchar(20) NOT NULL)

CREATE TABLE dbo.Vendor
   (VendorID                           int IDENTITY(1,1) PRIMARY KEY,
    Name                       varchar(80) NOT NULL,
    Address                    varchar(80) NOT NULL,
    Phone                      varchar(20) NOT NULL)

CREATE TABLE dbo.Product
   (ProductID                          int IDENTITY(1,1) PRIMARY KEY,
    VendorID                           int NOT NULL,
    Description                varchar(80) NOT NULL,
    WholesaleCost             decimal(7,2) NOT NULL,
    RetailPrice               decimal(7,2) NOT NULL)

CREATE TABLE dbo.Purchase
   (PurchaseID                         int IDENTITY(1,1) PRIMARY KEY,
    CustomerID                         int NOT NULL,
    OrderNumber                        int NOT NULL,
    OrderDate                smalldatetime NOT NULL)

CREATE TABLE dbo.PurchaseItem
   (PurchaseItemID                     int IDENTITY(1,1) PRIMARY KEY,
    PurchaseID                         int NOT NULL,
    ProductID                          int NOT NULL,
    Quantity                      smallint NOT NULL,
    LineNumber                    smallint NOT NULL)

CREATE TABLE dbo.Shipment
   (ShipmentID                         int IDENTITY(1,1) PRIMARY KEY,
    Carrier                    varchar(20) NOT NULL,
    TrackingNumber             varchar(20) NOT NULL,
    ShipDate                 smalldatetime NOT NULL)

CREATE TABLE dbo.PurchaseItemShipment
   (PurchaseItemShipmentID             int IDENTITY(1,1) PRIMARY KEY,
    PurchaseItemID                     int NOT NULL,
    ShipmentID                         int NOT NULL)

CREATE TABLE dbo.Payment
   (PaymentID                          int IDENTITY(1,1) PRIMARY KEY,
    PurchaseID                         int NOT NULL,
    Method                     varchar(20) NOT NULL,
    Amount                    decimal(7,2) NOT NULL)
```

Self-Referencing Tables:

A self-referencing table is a table that has a foreign key reference pointing to itself. The typical example is an Employee table where each record references another record in the same table as a supervisor. SQL Server does not prevent such a structure, but this database architecture strongly recommends against it. It causes problems when using generic routines to examine the database hierarchy. If a routine attempts to follow every reference then a self-referencing table causes an endless loop (circular reference). If a routine specifically ignores self-references then it fails to perform a complete analysis of table relationships. Both situations are very unacceptable and this database architecture strongly suggests that self-references never be used. They can be avoided with a fairly simple modification to the data model. The data model can use a set of three tables in place of a single Employee table that includes a self-reference. The three tables would be named Employee, Supervisor, and Position.

> The Employee table would have all the usual fields, much like the original table. It would contain one record for each employee.
>
> The Supervisor table would be a child of the Employee table. It would contain one record for each employee with supervisory responsibility.
>
> The Position table would be a child of the Employee table and the Supervisor table. It would contain at least one record for each employee, except the company president (who is the only employee without a supervisor). Each job position is filled by an employee who reports to a supervisor. One employee could fill several positions and could potentially report to several different supervisors through the positions.

This structure avoids the problems of a single self-referencing table, while it also provides much more flexibility. It allows for attributes that are unique to a supervisor, such as the effective date of the supervisory role or HR contact person information. It allows for attributes that are unique to a job position, such as the percentage of FTE necessary for the position or the location where work is performed. Open positions could be represented through an optional employee foreign key or by pointing the employee foreign key to a specially designated record.

A similar set of three tables could be used to model other relationships. Consider a machine parts business that sells individual items and sells assemblies of several items. The Employee table could become a Part table with many attributes. The Supervisor table could become an Assembly table with attributes such as who assembles the part or how much time it takes to assemble. The Position table could become a Component table with attributes such as the sequence of assembly or the quantity required by the assembly.

Indexes:

Indexes in a database are much like indexes in a book. They are used to quickly locate specific content. The proper use of indexes is probably the single most important factor in achieving the maximum level of performance from queries. Alas, there is no magic formula for doing proper indexing. It's heavily dependent on the data itself and how the data is being used.

This database architecture suggests a starting point for indexing where an index is created for every foreign key in every table. Such indexes will improve the performance of queries that select based on foreign keys. It also improves the performance of queries that include joins between tables, and a properly normalized database means lots of joins.

Once indexes have been created for all foreign keys, further indexing requires an understanding of the data itself and how the data is being used. Indexes are best applied to fields that are small, highly selective (many unique values), and often used in queries. In SQL Server, indexes can be clustered or not clustered. Only one index per table can be clustered, and that should be reserved for data that is frequently used in queries for aggregation (GROUP BY), sorting (ORDER BY), or filtering (WHERE). Indexes can also be applied to combinations of fields, in which case the most frequently referenced field(s) should be listed first. A query that references only the fields included in an index is especially efficient.

Be careful to avoid creating too many indexes. Indexes must be maintained by the system and that involves processing overhead during inserts and updates.

When experimenting with different indexes it's very important to carefully test the performance of existing queries. Do not be misled by the effects of data caching at the server.

Concurrency:

Any multi-user database application has to have some method of dealing with concurrent access to data. Concurrent access means that more than one user is accessing the same data at the same time. A problem occurs when user X reads a record for editing, user Y reads the same record for editing, user Y saves changes, then user X saves changes. The changes made by user Y are lost unless something prevents user X from blindly overwriting the record. This problem may not be handled by the database alone. The solution usually involves the application as well.

One way of dealing with concurrent access is to use a counter field that is incremented with each modification to the record. In the example above, user X would be alerted by the application that the record had been modified between the time of reading and the time of attempting to save changes. This situation is noticed by the database and/or by the application because the counter had changed. The application should allow user X to start over with the current record contents, or overwrite the changes made by user Y. No changes would be unknowingly lost.

Audit Trails:

Some businesses require that a certain amount of database change history be retained. Usually, such a requirement does not involve every table in the database. Maybe only selected tables need to have an audit trail (a list of changes for each record). It’s quite common to use triggers to implement a solution for this need. A trigger on each table senses when changes are being made and saves a copy of the changes in a corresponding change history table. This database architecture suggests a slightly different approach.

A trigger works very well to sense when changes are being made to a record, but it seems redundant to save every change as it’s being made (including inserts, updates, and deletes). Instead, save the previous record contents when a record is changed or deleted. This method requires less space because the current record contents are in only one place, the original table. The change history table (audit trail) contains only previous versions of each record, as well as the last version if the record has been deleted. A complete history for any particular record is available by listing the appropriate records from both tables.

Standard Fields:

This database architecture suggests up to six fields that could be included in every table. It's not likely that all six are needed for every database, but some are fairly universal. It would be best to use the same set of standard fields in every table. That's not a requirement, but consistency is a very desirable thing. Such consistency keeps open the options for dynamic routines that work for any table(s) in the database.

`CreateUser` or `strCreateUser` or `lngCreateUser`

This field would be used to identify the user that created the record. The field may contain a user name, or it may contain a foreign key if the application has internal security.

`CreateDate` or `dtmCreateDate`

This field would be used to store the date and time when the record was created.

`ModifyUser` or `strModifyUser` or `lngModifyUser`

This field would be used to identify the user that last modified the record. The field may contain a user name, or it may contain a foreign key if the application has internal security.

`ModifyDate` or `dtmModifyDate`

This field would be used to store the date and time when the record was last modified.

`DetectWork` or `lngDetectWork`

This field would be used to store a counter for detecting concurrent access to the record.

The database and application would need to work together to take advantage of this field…

The application reads a record for editing. It increments the counter field by one when saving changes. The database (a stored procedure) checks the stored counter before updating. If the stored counter is not less than the incremented counter, then a concurrency violation has taken place. In other words, another user changed the record between reading and attempting to save changes. The database denies the requested update and informs the application that the update was not performed. The application offers the user options for proceeding. If the user chooses to overwrite the record, the application simply increments the counter by one and tries to save again. As an alternative to incrementing the counter by one, the application could increment the counter by a very large number (such as a million) to indicate that a concurrency violation has taken place and to force the save to succeed. In this case, the counter would be used to track concurrency violations as well as updates.

`RecordMask` or `bytRecordMask`

This field would be used to store a status indicator for the record. In some situations, records in selected tables may have several different states of existence. This field could be used to mark records as active, inactive, pending, disabled, deleted, or any other state that business needs may dictate. If records would be marked as either active or deleted only, it may be better to consider record audit trail functionality (see above).

Standard Routines:

This database architecture suggests several standard routines for each table. Such routines can be generated from the database schema rather than hand-coded. Generated routines should be flexible for developers to use, but they must be carefully designed to minimize the occurrences of table scans (a common reason for performance issues) and to maximize the benefit of cached execution plans. Dynamic T-SQL is sometimes the best way to balance these needs.

Standard Triggers:

This database architecture suggests up to two standard triggers for each table. Each trigger simply sets one standard date/time field (if used) to the current date and time. If the standard stored procedures (see below) are used for all record inserts and updates, then these triggers are not required. They are mainly useful for handling the date/time fields when record inserts and updates are done by custom routines or by editing tools (such as Enterprise Manager).

`trgGRCITableName`

This trigger fires with an insert and sets the dtmCreateDate to the current date and time.

`trgGRCUTableName`

This trigger fires with an update and sets the dtmModifyDate to the current date and time.

Standard Stored Procedures:

This database architecture suggests up to six standard stored procedures for each table. They each perform one of the four basic SQL commands (INSERT, UPDATE, DELETE, SELECT).

`uspGRCITableName`

This routine performs an INSERT. It accepts values for each field (except for the primary key) and it returns the system-generated primary key (IDENTITY value) as an OUTPUT parameter.

This routine can automatically assign values to the Create and Modify standard fields.

`uspGRCUTableName`

This routine performs an UPDATE of one record. It accepts a primary key value and values for each of the other fields. If the DetectWork standard field is present, the new value is compared with the existing value. The existing value must be less than the new value or the record is not affected. The application can check the number of records affected and react accordingly.

This routine can automatically assign values to the Modify standard fields.

`uspGRCDTableName`

This routine performs a DELETE of one record.

This routine accepts (expects) a primary key value.

`uspGRCSTableName`

This routine performs a SELECT of one record. It returns all fields.

This routine accepts (expects) a primary key value.

`uspGRKDTableName`

This routine performs a DELETE of one or more records.

This routine accepts an optional field name to determine which key field will be used to select records. The default behavior is to use the primary key, but any foreign key can be used instead. This routine accepts an optional key value. The default behavior is to use a key value of zero (0), which matches all records. This routine also accepts an optional value for the RecordMask standard field. The default behavior is use a value of zero (0), which matches all records. If the RecordMask standard field is not present the value is simply ignored.

It's probably better to generate a separate stored procedure for each foreign key.

`uspGRKSTableName`

This routine performs a SELECT of one or more records. It returns all fields.

This routine accepts an optional field name to determine which key field will be used to select records. The default behavior is to use the primary key, but any foreign key can be used instead. This routine accepts an optional key value. The default behavior is to use a key value of zero (0), which matches all records. This routine also accepts an optional value for the RecordMask standard field. The default behavior is use a value of zero (0), which matches all records. If the RecordMask standard field is not present the value is simply ignored.

It's probably better to generate a separate stored procedure for each foreign key.

Standard User-Defined Function:

This database architecture suggests one standard user-defined function for each table. The function returns a set of records (all fields) with a single SELECT statement. Such a function offers advantages over a view or stored procedure. It allows the use of parameters (a view does not) and it can be referenced in a FROM clause (unlike a stored procedure).

`udfGRKSTableName`

This function performs a SELECT of one or more records. It returns all fields.

This function expects a field name to determine which key field will be used to select records. The primary key or any foreign key can be used. This function expects a key value to be supplied. The value zero (0) matches all records. This function also expects a value for the RecordMask standard field as a way to filter or mask records based on their status. The value zero (0) matches all records. If the RecordMask standard field is not present the value is simply ignored. The parameter is included for all tables for consistency.

Standard View:

This database architecture suggests one standard view for each table. The view returns all the records in the table with a single SELECT statement. An ORDER BY clause is included to sort the records by the primary key (or the first column of a clustered index).

```
qryGROSTableName
```

This view performs a SELECT of all records. It returns all fields.

This view orders the records by the primary key, or the first column of a clustered index if such an index has been created.

The standard view is no longer suggested. It does not serve a useful purpose.

Generated Middle-Tier Code:

The structure of this database architecture allows standard database access code to be generated by a routine, instead of having to enter it by endless typing.

The generated code could handle the CRUD (Create, Retrieve, Update, Delete) operations for single records in many tables of most databases.

Here are two ideas for code that could be generated according to the database schema…

Generate a class module for each table. The class module could have methods for the CRUD operations and properties for the fields.

Generate a web page or Windows form for each table. The page/form could provide the user interface for the CRUD operations.

Bulk Relational Data:

A relational database management system (RDBMS), such as SQL Server, is intended to handle relational data. For the purpose of this discussion, relational data is defined as data with a strict hierarchical organization or tree structure. The RDBMS ensures the integrity of relational data by enforcing various constraints. The constraints…

* guarantee that primary key values are unique
* prevent child records from being inserted if they reference parent records that do not exist
* prevent parent records from being deleted if there are child records referring to them

Bulk relational data often consists of a set of related records that come from a group of related tables. The tables represent a logical subset of the complete data model and they form a branch of the database tree (hierarchy). A common example might include one record from a Customer table, all the child records of the customer from an Order table, and all the child records of those orders from an OrderDetail table. It may be necessary to manipulate a set of records like this as a unit. SQL Server 2000 offers a cascading delete feature for removing a set of related records as a unit, but copying the records is a much more involved operation.

The order in which tables are processed when handling bulk relational data is critical. Deleting such data is not difficult, so this section focuses on the insertion of such data.

The handling of bulk relational data is one situation (perhaps the only situation) where natural data primary keys and foreign keys have an advantage. With natural data keys the only issue is processing the tables in the correct order. The keys have meaning outside the database and they are contained within the bulk data. New child records can be inserted as they are, without any need to modify foreign keys to match the primary keys in corresponding new parent records.

The handling of bulk relational data in this database architecture can be quite involved. With surrogate keys, foreign keys in new child records must be adjusted to reference the generated primary keys in corresponding new parent records. Making the adjustments before insertion usually requires row-by-row processing using complex code. Making the adjustments after insertion requires disabling foreign key constraints and executing some very intricate code.

The well-known SQL Server utility products provide no assistance with the handling of bulk relational data containing surrogate keys. The tools that generate INSERT statement scripts do nothing to modify foreign keys in new child records. However, the consistency of this database architecture makes it feasible to write generic routines that handle bulk relational data as a unit. A unit could be copied to a demonstration database or a development database. A unit could be copied to an archive database and removed from a production database. A unit could be copied within a production database to avoid manual data entry.

General-Purpose Routines:

One of the great benefits of using this database architecture is that it’s compatible with creating many powerful generic routines to handle common DBA chores.

Here are some very handy things that T-SQL utility routines could do to implement and support this database architecture (see example code below)…

Create a script for the database schema, adding selected standard fields, naming foreign keys appropriately, and applying correct prefixes to field names.

Create a script for the standard triggers, stored procedures, user-defined function, and view.

Create (or delete) constraints that establish primary keys.

Create (or delete) foreign key constraints.

Create (or delete) foreign key indexes.

Find duplicate records in a table.

Given a record in a table, count the associated records in all child tables.

Compare records in a table with records in a corresponding table in another database.

INSERT/UPDATE/DELETE records in a table in order to make it match a corresponding table in another database.

Delete a branch of a database, a branch being a set of records in a table and all the descendent records in other tables.

Copy a branch of a database to another database (with the same schema), adjusting the foreign keys in all the copied records.

Given a record in a table and a set of related tables, return a set of associated records from each table. This returns much less data than the usual (denormalized) view.

The following T-SQL code references system tables. The use of properly documented (BOL) system table columns should be safe and reliable for most database administration purposes, but it may be best to avoid the practice in production application code.

The code below creates primary keys and foreign keys for this database architecture. It's a very simple example of a general-purpose routine.

The first page (this page) is initialization.

The second page creates primary keys.

The third page creates foreign keys.

```
DECLARE @Return int
DECLARE @Retain int
DECLARE @Status int

SET @Status = 0

DECLARE @Prefix varchar(10)

DECLARE @Length tinyint
DECLARE @Locate tinyint

SET @Prefix = 'tbl' -- table name prefix

SET @Length = LEN(@Prefix)
SET @Locate = LEN(@Prefix) + 1

DECLARE @RunSQL varchar(2000)

DECLARE @Object varchar(100)
DECLARE @Column varchar(100)
DECLARE @Parent varchar(100)

DECLARE @Format varchar(100)

SET @Format = 'lng*ID'
```

```
-- Create Primary Keys

 DECLARE Tables CURSOR FAST_FORWARD FOR
  SELECT O.name, C.name
    FROM sysobjects AS O
    JOIN syscolumns AS C
      ON O.id = C.id
   WHERE ISNULL(OBJECTPROPERTY(O.id,'IsMSShipped'),1) = 0
     AND RTRIM(O.type) = 'U'
     AND LEFT(O.name,@Length) = @Prefix
     AND O.name NOT LIKE '%dtproper%'
     AND O.name NOT LIKE 'dt[_]%'
     AND C.colid = 1
ORDER BY O.name

OPEN Tables

SET @Retain = @@ERROR IF @Status = 0 SET @Status = @Retain

FETCH NEXT FROM Tables INTO @Object, @Column

SET @Retain = @@ERROR IF @Status = 0 SET @Status = @Retain

WHILE @@FETCH_STATUS = 0 AND @Status = 0

    BEGIN

    SET @RunSQL = ' ALTER TABLE ' + @Object
                + ' ADD CONSTRAINT ' + 'keyPK'
                + SUBSTRING(@Object,@Locate,100)
                + ' PRIMARY KEY NONCLUSTERED'
                + ' (' + @Column + ')'

    IF @Status = 0 EXECUTE (@RunSQL) SET @Return = @@ERROR

    IF @Status = 0 SET @Status = @Return

    FETCH NEXT FROM Tables INTO @Object, @Column

    SET @Retain = @@ERROR IF @Status = 0 SET @Status = @Retain

    END

CLOSE Tables DEALLOCATE Tables
```

```
-- Create Foreign Keys

 DECLARE Tables CURSOR FAST_FORWARD FOR
  SELECT O.name, C.name, T.name
    FROM sysobjects AS O
    JOIN syscolumns AS C
      ON O.id = C.id
    JOIN sysobjects AS T
      ON C.name = REPLACE(@Format,'*',SUBSTRING(T.name,@Locate,100))
     AND C.colid > 1
   WHERE ISNULL(OBJECTPROPERTY(O.id,'IsMSShipped'),1) = 0
     AND RTRIM(O.type) = 'U'
     AND LEFT(O.name,@Length) = @Prefix
     AND O.name NOT LIKE '%dtproper%'
     AND O.name NOT LIKE 'dt[_]%'
     AND ISNULL(OBJECTPROPERTY(T.id,'IsMSShipped'),1) = 0
     AND RTRIM(T.type) = 'U'
     AND LEFT(T.name,@Length) = @Prefix
     AND T.name NOT LIKE '%dtproper%'
     AND T.name NOT LIKE 'dt[_]%'
ORDER BY O.name

OPEN Tables

SET @Retain = @@ERROR IF @Status = 0 SET @Status = @Retain

FETCH NEXT FROM Tables INTO @Object, @Column, @Parent

SET @Retain = @@ERROR IF @Status = 0 SET @Status = @Retain

WHILE @@FETCH_STATUS = 0 AND @Status = 0

    BEGIN

    SET @RunSQL = ' ALTER TABLE ' + @Object
                + ' ADD CONSTRAINT ' + 'keyFK'
                + SUBSTRING(@Object,@Locate,100)
                + SUBSTRING(@Parent,@Locate,100)
                + ' FOREIGN KEY'
                + ' (' + @Column + ')'
                + ' REFERENCES ' + @Parent
                + ' (' + @Column + ')'

    IF @Status = 0 EXECUTE (@RunSQL) SET @Return = @@ERROR

    IF @Status = 0 SET @Status = @Return

    FETCH NEXT FROM Tables INTO @Object, @Column, @Parent

    SET @Retain = @@ERROR IF @Status = 0 SET @Status = @Retain

    END

CLOSE Tables DEALLOCATE Tables
```

T-SQL Coding Standards:

Coding standards are often overlooked when it comes to T-SQL programming. However, they are very important to a smoothly operating development team. The coding standards presented here are not universally accepted, there is no such thing. Some of these suggestions are rather subjective and other SQL Server shops may have different ideas that are equally as effective. The most important thing is establishing reasonable coding standards and adhering to them.

The remainder of this section contains miscellaneous coding standards, hints, and tips for T-SQL programming. The comments are not necessarily in a suggested order of priority or importance. This section is not intended to be a resource for learning basic T-SQL syntax. BOL should be considered a valuable reference for finding and studying T-SQL syntax examples.

On the surface, the formatting of T-SQL code may not seem very important. However, uniform formatting of T-SQL code is extremely helpful in understanding each other's work. Statements follow a certain structure, and having that structure be visually evident makes it much easier to verify that various parts of the statements (tables, fields, joins, conditions) are correct. Uniform formatting also makes it much easier to add/remove sections to/from complex T-SQL statements for debugging purposes. Here's a formatting example for a SELECT statement…

```
   SELECT C.Name
        , E.NameLast
        , E.NameFirst
        , E.Number
        , ISNULL(I.Description,'NA') AS Description
     FROM tblCompany   AS C
     JOIN tblEmployee  AS E
       ON C.CompanyID
        = E.CompanyID
LEFT JOIN tblCoverage  AS V
       ON E.EmployeeID
        = V.EmployeeID
LEFT JOIN tblInsurance AS I
       ON V.InsuranceID
        = I.InsuranceID
    WHERE C.Name LIKE @Name
      AND V.CreateDate > CONVERT(smalldatetime,'01/01/2000')
 ORDER BY C.Name
        , E.NameLast
        , E.NameFirst
        , E.Number
        , ISNULL(I.Description,'NA')
```

Use four-space indentation for statements within a nested block of code (a multi-line SELECT statement like the example above is still a single SQL statement). Right-align SQL keywords when starting a new line within the same statement. Configure the code editor to use spaces instead of tabs. This makes the formatting appear consistent regardless of what application anybody else uses to view the code (with a fixed-width font).

Capitalize all T-SQL keywords, including T-SQL functions. Use mixed-case for variable names and cursor names. Use lower-case for data types (as in DECLARE or CREATE).

Keep table name aliases short, but as meaningful as possible. In general, use the capital letters of a table name as an alias. Use the AS keyword to specify aliases for tables or fields.

Always qualify field names with table name aliases when there are multiple tables involved in a T-SQL statement. This adds clarity for others and avoids ambiguous references.

Align related numbers into columns when they appear in contiguous lines of code (such as with a series of SUBSTRING function calls). This makes the list of numbers easier to scan.

Use single blank lines to separate logical pieces of T-SQL code, and do so liberally. Do not use double blank lines. These things make the code easier to read for others.

As with any programming language, the use of local variables in T-SQL (such as @lngTableID) is a major part of the coding. That by itself makes their proper use an important factor in writing clean code that others can interpret as quickly as possible. Using variables properly also tends to promote more robust coding with fewer bugs. Be sure to use appropriate data type declarations, correct data type assignments, and consistent capitalization.

Always specify the length of a character data type. Be sure to allow for the maximum number of characters. Characters beyond the maximum length are simply lost.

Always specify the precision and scale of the decimal data type. The default precision is 18 and the default scale is 0 (resulting in an integer).

The error-checking examples in BOL do not work as written! The T-SQL statement (IF) used to check the @@ERROR system function actually clears the @@ERROR value in the process. It would never capture a value other than zero. Even if the examples worked, they would capture the last error that occurred. It would often be more helpful to capture the first error. The error code must be captured immediately using SET or SELECT (see the assorted examples above). The error code should then be transferred to a status variable if the status variable is still zero (which indicates no error has occurred). This method reliably captures the first error.

Be careful with mathematical formulas. T-SQL may force an expression into an unintended data type. Add point zero (.0) to integer constants if a decimal result is desired.

Do not use the STR function to perform any rounding, use it with integer values only. Use the CONVERT function (going to a different scale) or the ROUND function before converting to a string if the string form of a decimal value is needed.

Do not rely upon any implicit data type conversions. For example, do not assign a character value to a numeric variable assuming that T-SQL will do the necessary conversion. Instead, always use the appropriate CONVERT function to make the data types match before doing a variable assignment or a value comparison.

Be aware that T-SQL appears to do an implicit and automatic RTRIM on character expressions before doing a comparison. However, do not depend upon that behavior.

Do not directly compare a variable value to NULL with comparison operators (symbols). If the variable value could be null, use IS NULL, IS NOT NULL, or the ISNULL function to perform the comparison. These methods are not subject to SQL Server settings.

Do not use double quotes in T-SQL code for string constants or identifiers. The meaning of double quotes is subject to SQL Server settings. Use single quotes for string constants. If it’s necessary to delimit an object name, use brackets around the name.

Do not rely upon any undocumented fields in system tables or any undocumented functionality of T-SQL statements. Things are undocumented if they are not described in BOL.

Never depend on a SELECT statement returning records in any particular order unless the order is specified with an ORDER BY clause.

In general, use an ORDER BY clause with a SELECT statement, especially during development or debugging. A known order, even if not the most convenient order, is better than no order for visually scanning the data. In some situations the order of the resulting records does not really matter. In those cases be sure to remove the ORDER BY clause before deploying the routine, thereby avoiding the additional overhead of sorting the records.

The use of an asterisk in the select list (SELECT * FROM) should generally be avoided when accessing typical tables. It would be unusual for all fields to be required for a given task, and returning any extra fields adds overhead. It may be quite appropriate to use an asterisk when accessing a temporary table, view, or table-valued user-defined function that was specifically designed for a given task. In those cases, using an asterisk in the select list means fewer code modifications if the task changes in the future.

Be cautious with table-valued user-defined functions. Passing in a parameter with a variable rather than a constant can result in table scans if the parameter is used in a WHERE clause. However, be aware that table-valued user-defined functions have some dynamic compilation features that make them very advantageous for optional parameters.

Avoid using the same table-valued user-defined function more than once in a single query, particularly if passing the same parameters each time.

Always look for a set-based solution to a problem before implementing a temporary table approach or a cursor-based approach. A set-based approach is often more efficient.

Consider using table variables in place of temporary tables with SQL Server 2000. They can avoid some resource contention, but be aware that indexing is very limited. If a table variable could contain a large set of data then experiment with a temporary table and indexing.

Create temporary tables early in the routine, and explicitly drop them at the end. Mixing DDL statements with DML statements can contribute to excessive recompile activity.

Temporary tables are not inherently evil. Using them appropriately can make some routines much more efficient. If a temporary table will be used only once after being populated, then possibly a derived table would be a better choice. However, if appropriate indexing is done, a temporary table may provide better performance than a derived table. If a certain set of data (particularly data from large tables) will be referenced repeatedly within a routine, then it’s usually beneficial to stash the data in a temporary table. Experiment with some different approaches to see which one offers the best performance for a given task.

Cursors are not inherently evil. A FAST_FORWARD cursor over a small set of data will often outperform other methods of handling row-by-row processing. This is especially true if several tables must be referenced to get the necessary data. A routine that includes “running totals” in a result set often executes fastest when implemented using a cursor. If development time allows it, try both a cursor-based approach and a set-based approach to see which one performs best.

Dynamic SQL is not inherently evil. It should not be used as a substitute for well-written stored procedures in a production environment, but it can be extremely handy if it’s used sparingly and appropriately. Some T-SQL statements and functions do not allow variables for their arguments. Dynamic SQL provides a way to execute such code using the desired parameters. For example, a BULK INSERT statement that imports a text file from a variable UNC path into a temporary table can be handled this way. Dynamic SQL can have some security ramifications because it requires that users have permission to directly access any permanent tables that are referenced. Dynamic SQL is especially useful for administrative tasks. It can be used to create some very powerful routines for database administration if the architecture is sufficiently consistent.

A table of sequence numbers, 1 through N, can be very handy for many purposes (see T-SQL Code Examples for a way to create such a table).

Understand how a CROSS JOIN works and take advantage of it. Frequently, a CROSS JOIN is used between a table of working data and a table of sequence numbers. The result set contains a row for every combination of working data and sequence number.

Avoid doing a JOIN between a local table and a remote table on a linked server. Such a query often provides very poor performance unless the remote table is small. Consider the available alternatives that transmit a filtered result set from the remote server to the local server. Some possible options are remote views, remote stored procedures, or the OPENQUERY function. The objective is to try to minimize the amount of data being transmitted between servers by performing as much of the query as possible on the remote server if that server contains the largest tables involved in the query.

Almost every stored procedure should have SET NOCOUNT ON near the beginning. It's good form to have SET NOCOUNT OFF near the end. This standard also applies to triggers.

An explicit transaction is suggested any time more than one database modification statement is used in a routine. This includes a single statement executed multiple times in a loop.

If several stored procedures are used to modify a common group of tables be sure to process the tables in the same order within each stored procedure. Also, keep the scope of each transaction as narrow as possible by doing any preliminary work (such as loading temporary tables) before starting the transaction. These things reduce the opportunity for a deadlock situation.

This section mentions some syntax that is specific to T-SQL. If code portability is a concern it may be advisable to use standard SQL syntax. As examples, the CAST function could be used instead of the CONVERT function for most data type conversions and the COALESCE function could be used instead of the ISNULL function.

The comments in this document that pertain to BOL were originally written when the current version of BOL was the version that came with the initial release of SQL Server 2000. More recent versions of BOL have been updated in many places. It's possible that comments in this document could become outdated as BOL changes.

T-SQL Code Examples:

Here are some simple T-SQL code examples in the form of user-defined functions. The code addresses several of the most common requests in SQL Server forums.

```
-- GPFillZeros converts an integer value to a zero-filled string of a given length

CREATE FUNCTION dbo.udfGPFillZeros
   (@N      int,
    @I  tinyint)
RETURNS varchar(10)
AS
BEGIN

DECLARE @Result varchar(10)

SET @Result = RIGHT(CONVERT(varchar(12),@N+10000000000),@I)

RETURN (@Result)
END

-- GPFixNumber converts a decimal value to a space-filled string of a given format

CREATE FUNCTION dbo.udfGPFixNumber
   (@N  decimal(19,6),
    @I  tinyint,
    @D  tinyint)
RETURNS varchar(20)
AS
BEGIN

DECLARE @Result varchar(20)

SET @Result = LEFT(RIGHT(SPACE(12)+CONVERT(varchar(20),ROUND(@N,@D)),@I+7),@I+@D+1)

RETURN (@Result)
END

-- GPGetSeries returns a table of integer sequence numbers of a given range

CREATE FUNCTION dbo.udfGPGetSeries
   (@A int,
    @Z int)
RETURNS @T TABLE (Number int)
AS
BEGIN

DECLARE @I int

SET @I = 1

INSERT @T VALUES (@A)

WHILE @I < @Z - @A + 1

    BEGIN

    INSERT @T
    SELECT @I + Number
      FROM @T
     WHERE @I + Number <= @Z

    SET @I = @I + @@ROWCOUNT

    END

RETURN
END
```

```
-- GPParseText returns a table of strings by parsing a given string using a given delimiter

CREATE FUNCTION dbo.udfGPParseText
   (@S varchar(7998),
    @E    char(   1))
RETURNS @T TABLE (Number smallint IDENTITY(1,1), Offset smallint, String varchar(2000))
AS
BEGIN

DECLARE @N smallint

DECLARE @Z varchar(8000)

SET @N = LEN(@S) + 1

SET @Z = @E + @S + @E

  INSERT @T (Offset, String)
  SELECT Number
       , SUBSTRING(@S,Number,CHARINDEX(@E,@Z,Number+1)-Number-1)
    FROM dbo.udfGPGetSeries(1,@N)
   WHERE SUBSTRING(@Z,Number,1) = @E
ORDER BY Number

RETURN
END

-- GPCountHits returns a count of how many times a search string is found in a given string

CREATE FUNCTION dbo.udfGPCountHits
   (@S varchar(8000),
    @E varchar(  80))
RETURNS smallint
AS
BEGIN

DECLARE @Result smallint

DECLARE @N smallint

SET @N = DATALENGTH(@S)

SELECT @Result = COUNT(*)
  FROM udfGPGetSeries(1,@N) AS T
 WHERE SUBSTRING(@S,T.Number,@N) LIKE @E + '%'

RETURN (@Result)
END
```

```
-- Examples

DECLARE @W int

SET @W = 123

SELECT dbo.udfGPFillZeros(@W,7)

-- Result = 0000123

DECLARE @M decimal(7,2)

SET @M = 8300

SELECT '$' + dbo.udfGPFixNumber(@M/800,5,2)

-- Result = $   10.38

DECLARE @X smallint
DECLARE @Y smallint

SET @X = 3
SET @Y = 8

SELECT * FROM dbo.udfGPGetSeries(@X,@Y)

-- Result = six rows, one column each, values 3 4 5 6 7 8

DECLARE @S varchar(2000)

SET @S = '1,12,123,1234,ABCD,ABC,AB,A,*'

SELECT * FROM dbo.udfGPParseText(@S,',')

-- Result = nine rows, three columns each, number/offset/string

DECLARE @S varchar(2000)

SET @S = 'football basketball baseball hockey soccer volleyball golf tennis'

SELECT dbo.udfGPCountHits(@S,'ball')

-- Result = 4

DECLARE @S varchar(2000)

SET @S = 'This fine line is a baseline for the incline of the main line.'

SELECT dbo.udfGPCountHits(@S,'[ ]line[;,.!? ]')

-- Result = 2
```

The following T-SQL code references system tables. The use of properly documented (BOL) system table columns should be safe and reliable for most database administration purposes, but it may be best to avoid the practice in production application code.

The code below returns a table of objects in the database. The table has fields for object type, parent object name (if applicable), object name, and object creation date (if available). The code provides parameters to include or exclude certain object types from the list. The object types are briefly defined within the code, but BOL includes a full description. The code can be used as a starting point to build routines that provide much more detailed information.

```
DECLARE @Return int
DECLARE @Retain int
DECLARE @Status int

SET @Status = 0

DECLARE @Prefix varchar(10)

DECLARE @Length tinyint
DECLARE @Locate tinyint

SET @Prefix = 'tbl' -- table name prefix

SET @Length = LEN(@Prefix)
SET @Locate = LEN(@Prefix) + 1

DECLARE @icrdate datetime

DECLARE @itype char(2)

SET @itype = 'I'

DECLARE @Include varchar(100)
DECLARE @Exclude varchar(100)

-- V  = View
-- P  = Stored Procedure
-- FN = User-Defined Function, Scalar
-- IF = User-Defined Function, Table-Valued, Inline
-- TF = User-Defined Function, Table-Valued, Multi-Statement
-- TR = Trigger
-- U  = Table
-- K  = Key, Primary OR Constraint, Unique
-- F  = Key, Foreign
-- C  = Constraint, Check
-- D  = Default
-- I  = Index (fake type)

-- SET @Include = 'V|P|FN|IF|TF|TR'

-- SET @Include = 'U|K|F|C|D|' + RTRIM(@itype)

-- SET @Exclude = 'U|K|F|C|D|' + RTRIM(@itype)
```

```
   SELECT        O.type     AS Type  ,
          ISNULL(T.name,'') AS Parent,
                 O.name     AS Object,
          ISNULL(CONVERT(varchar(30),O.crdate,120),'') AS Creation
     FROM sysobjects AS O
LEFT JOIN sysobjects AS T
       ON O.parent_obj = T.id
    WHERE ISNULL(OBJECTPROPERTY(O.id,'IsMSShipped'),1) = 0
      AND O.name NOT LIKE '%dtproper%'
      AND O.name NOT LIKE 'dt[_]%'
      AND (@Include IS NULL
       OR  CHARINDEX('|'+RTRIM(O.type)+'|','|'+(@Include)+'|') > 0)
      AND (@Exclude IS NULL
       OR  CHARINDEX('|'+RTRIM(O.type)+'|','|'+(@Exclude)+'|') = 0)

    UNION ALL

   SELECT        @itype     AS Type  ,
          ISNULL(O.name,'') AS Parent,
                 I.name     AS Object,
          ISNULL(CONVERT(varchar(30),@icrdate,120),'') AS Creation
     FROM sysobjects AS O
     JOIN sysindexes AS I
       ON O.id = I.id
    WHERE ISNULL(OBJECTPROPERTY(O.id,'IsMSShipped'),1) = 0
      AND RTRIM(O.type) = 'U'
      AND LEFT(O.name,@Length) = @Prefix
      AND O.name NOT LIKE '%dtproper%'
      AND O.name NOT LIKE 'dt[_]%'
      AND I.indid BETWEEN 1 AND 254
      AND LEFT(I.name,8) <> '_WA_Sys_'
      AND (@Include IS NULL
       OR  CHARINDEX('|'+RTRIM(@itype)+'|','|'+(@Include)+'|') > 0)
      AND (@Exclude IS NULL
       OR  CHARINDEX('|'+RTRIM(@itype)+'|','|'+(@Exclude)+'|') = 0)

 ORDER BY Type, Parent, Object

SET @Retain = @@ERROR IF @Status = 0 SET @Status = @Retain
```

Conclusion:

As stated in the introduction, this database architecture may not be suitable exactly as-is for every situation. It provides an excellent foundation for a tailored solution that meets specific needs. It also points out several important considerations that need to be addressed as part of such a solution. There are many technical issues to think about, and the design decisions made early in the process can have a tremendous long-term impact.

The Appendix contains a database diagram and some example T-SQL script. They serve to illustrate and demonstrate this database architecture. The script creates tables, primary keys, foreign keys, indexes, and selected objects for a very simplistic insurance claims processing company. All of the T-SQL script here was generated rather than typed.

A large package of T-SQL routines that implements and supports this database architecture is available from the author. The diverse collection provides a wealth of functionality.

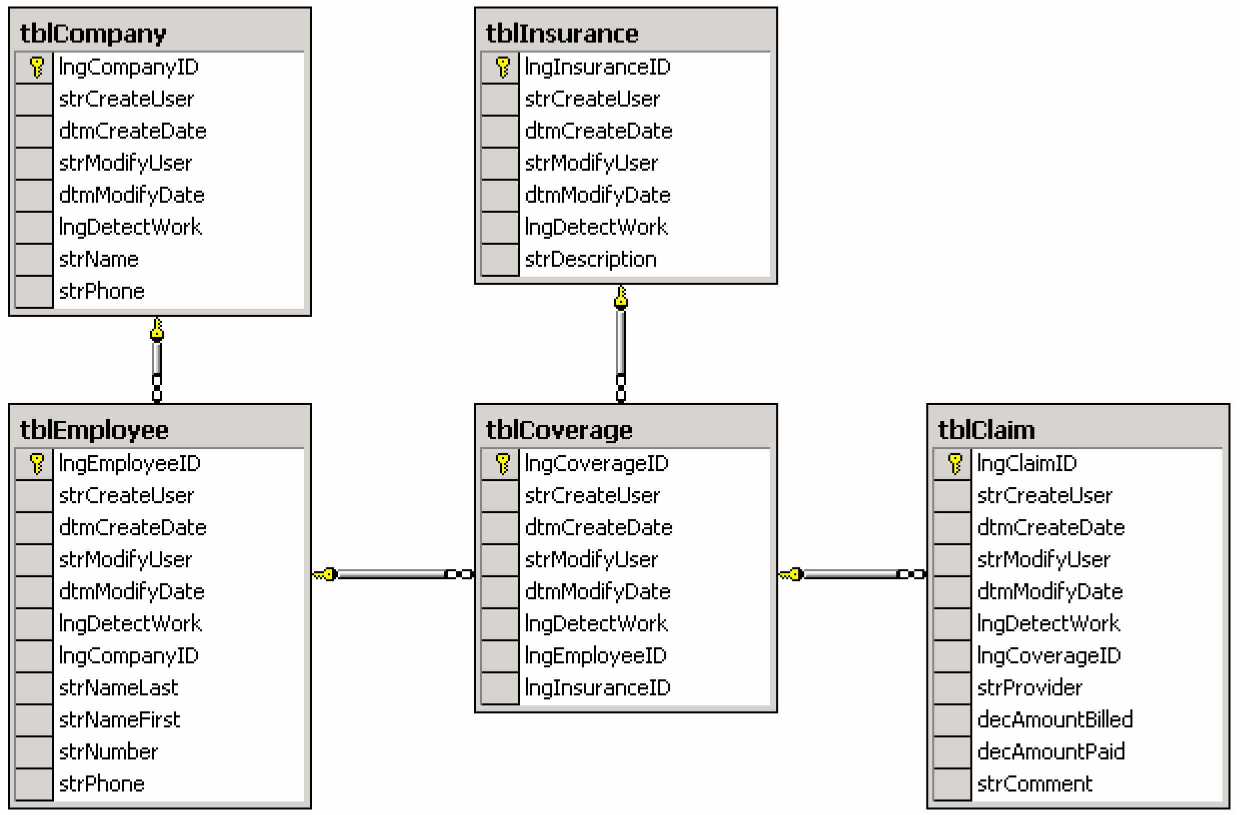
A family of utility applications for SQL Server is also available. The tools provide many very powerful features for those who prefer the convenience of a graphical user interface.

For more information, please visit http://www.wingenious.com.

Wingenious

… ingenious utility software solutions for Windows 95/98/Me/NT/2000/XP, Office 97/2000, SQL Server 7/2000

Appendix:



```
CREATE TABLE dbo.tblInsurance
   (lngInsuranceID                          int IDENTITY(1,1),
    strCreateUser                   varchar(40)     NULL,
    dtmCreateDate                 smalldatetime     NULL,
    strModifyUser                   varchar(40)     NULL,
    dtmModifyDate                 smalldatetime     NULL,
    lngDetectWork                           int NOT NULL,
    strDescription                  varchar(40) NOT NULL)

CREATE TABLE dbo.tblCompany
   (lngCompanyID                            int IDENTITY(1,1),
    strCreateUser                   varchar(40)     NULL,
    dtmCreateDate                 smalldatetime     NULL,
    strModifyUser                   varchar(40)     NULL,
    dtmModifyDate                 smalldatetime     NULL,
    lngDetectWork                           int NOT NULL,
    strName                         varchar(40) NOT NULL,
    strPhone                        varchar(14) NOT NULL)
```

```
CREATE TABLE dbo.tblEmployee
   (lngEmployeeID                                        int IDENTITY(1,1),
    strCreateUser                                varchar(40)     NULL,
    dtmCreateDate                              smalldatetime     NULL,
    strModifyUser                                varchar(40)     NULL,
    dtmModifyDate                              smalldatetime     NULL,
    lngDetectWork                                        int NOT NULL,
    lngCompanyID                                         int NOT NULL,
    strNameLast                                  varchar(30) NOT NULL,
    strNameFirst                                 varchar(24) NOT NULL,
    strNumber                                    varchar(20) NOT NULL,
    strPhone                                     varchar(14) NOT NULL)

CREATE TABLE dbo.tblCoverage
   (lngCoverageID                                        int IDENTITY(1,1),
    strCreateUser                                varchar(40)     NULL,
    dtmCreateDate                              smalldatetime     NULL,
    strModifyUser                                varchar(40)     NULL,
    dtmModifyDate                              smalldatetime     NULL,
    lngDetectWork                                        int NOT NULL,
    lngEmployeeID                                        int NOT NULL,
    lngInsuranceID                                       int NOT NULL)

CREATE TABLE dbo.tblClaim
   (lngClaimID                                           int IDENTITY(1,1),
    strCreateUser                                varchar(40)     NULL,
    dtmCreateDate                              smalldatetime     NULL,
    strModifyUser                                varchar(40)     NULL,
    dtmModifyDate                              smalldatetime     NULL,
    lngDetectWork                                        int NOT NULL,
    lngCoverageID                                        int NOT NULL,
    strProvider                                  varchar(40) NOT NULL,
    decAmountBilled                             decimal(9,2) NOT NULL,
    decAmountPaid                               decimal(9,2) NOT NULL,
    strComment                                 varchar(2000) NOT NULL)
```

```
ALTER TABLE tblInsurance
    ADD CONSTRAINT keyPKInsurance
    PRIMARY KEY NONCLUSTERED (lngInsuranceID)

ALTER TABLE tblCompany
    ADD CONSTRAINT keyPKCompany
    PRIMARY KEY NONCLUSTERED (lngCompanyID)

ALTER TABLE tblEmployee
    ADD CONSTRAINT keyPKEmployee
    PRIMARY KEY NONCLUSTERED (lngEmployeeID)

ALTER TABLE tblCoverage
    ADD CONSTRAINT keyPKCoverage
    PRIMARY KEY NONCLUSTERED (lngCoverageID)

ALTER TABLE tblClaim
    ADD CONSTRAINT keyPKClaim
    PRIMARY KEY NONCLUSTERED (lngClaimID)


ALTER TABLE tblEmployee
    ADD CONSTRAINT keyFKEmployeeCompany
    FOREIGN KEY (lngCompanyID)
    REFERENCES tblCompany (lngCompanyID)

ALTER TABLE tblCoverage
    ADD CONSTRAINT keyFKCoverageInsurance
    FOREIGN KEY (lngInsuranceID)
    REFERENCES tblInsurance (lngInsuranceID)

ALTER TABLE tblCoverage
    ADD CONSTRAINT keyFKCoverageEmployee
    FOREIGN KEY (lngEmployeeID)
    REFERENCES tblEmployee (lngEmployeeID)

ALTER TABLE tblClaim
    ADD CONSTRAINT keyFKClaimCoverage
    FOREIGN KEY (lngCoverageID)
    REFERENCES tblCoverage (lngCoverageID)
```

```
CREATE INDEX idxCompany
    ON tblEmployee (lngCompanyID)
    WITH FILLFACTOR = 90

CREATE INDEX idxInsurance
    ON tblCoverage (lngInsuranceID)
    WITH FILLFACTOR = 90

CREATE INDEX idxEmployee
    ON tblCoverage (lngEmployeeID)
    WITH FILLFACTOR = 90

CREATE INDEX idxCoverage
    ON tblClaim (lngCoverageID)
    WITH FILLFACTOR = 90


IF EXISTS (SELECT * FROM sysobjects WHERE name = 'trgGRCICompany')
    DROP TRIGGER trgGRCICompany
GO
CREATE TRIGGER trgGRCICompany ON tblCompany
FOR INSERT
AS
SET NOCOUNT ON
UPDATE tblCompany SET dtmCreateDate = GETDATE()
  FROM tblCompany AS T JOIN Inserted AS I ON T.lngCompanyID = I.lngCompanyID
SET NOCOUNT OFF
GO

IF EXISTS (SELECT * FROM sysobjects WHERE name = 'trgGRCUCompany')
    DROP TRIGGER trgGRCUCompany
GO
CREATE TRIGGER trgGRCUCompany ON tblCompany
FOR UPDATE
AS
SET NOCOUNT ON
UPDATE tblCompany SET dtmModifyDate = GETDATE()
  FROM tblCompany AS T JOIN Inserted AS I ON T.lngCompanyID = I.lngCompanyID
SET NOCOUNT OFF
GO

IF EXISTS (SELECT * FROM sysobjects WHERE name = 'qryGROSCompany')
    DROP VIEW dbo.qryGROSCompany
GO
CREATE VIEW dbo.qryGROSCompany
AS
  SELECT TOP 100 PERCENT WITH TIES *
    FROM tblCompany WITH (NOLOCK)
ORDER BY lngCompanyID
GO
```

```
IF EXISTS (SELECT * FROM sysobjects WHERE name = 'uspGRCDCompany')
    DROP PROCEDURE dbo.uspGRCDCompany
GO
CREATE PROCEDURE dbo.uspGRCDCompany
    @DBValue     int,
    @DBAdmin tinyint = NULL
AS
DELETE tblCompany
 WHERE lngCompanyID = @DBValue
RETURN (@@ERROR)
GO

IF EXISTS (SELECT * FROM sysobjects WHERE name = 'uspGRCSCompany')
    DROP PROCEDURE dbo.uspGRCSCompany
GO
CREATE PROCEDURE dbo.uspGRCSCompany
    @DBValue     int,
    @DBAdmin tinyint = NULL
AS
SELECT *
  FROM tblCompany WITH (NOLOCK)
 WHERE lngCompanyID = @DBValue
RETURN (@@ERROR)
GO
```

Thanks for reading the Database Architecture document!

The document stressed several key points, with the most important being consistency in database architecture. It also focused on complete Declarative Referential Integrity (DRI), which includes primary key constraints and foreign key constraints. Attention was given to indexing and taking advantage of the power of code generation. Wingenious encountered a need to ensure consistency, complete DRI, efficient indexing, and robust routines in every database it built and maintained. This critical task could not be done thoroughly by using manual inspection, so Wingenious designed and developed a software tool.

The tool is called DBGizmo, and it's incredibly useful for the purposes listed above. The application has 17 tabs, with the first seven tabs (Home, Tables, Routines, SQL^2, SQL^3, Notes, Facts) containing the key features. The key features provide numerous ways to analyze databases and generate powerful code for working with databases.

The Home tab provides many searching and scripting features, as well as quick access to three lists of 10 (each) query data sets for database analysis.

The Tables tab provides convenient ways to browse table relationships and view column definitions or associated objects.

The Routines tab provides convenient ways to review SQL routine dependencies and the SQL code definitions.

The SQL^2 tab provides several options to generate handy SQL code for a wide variety of mostly DDL operations.

The SQL^3 tab provides two options to generate extremely powerful SQL code for various common DML operations.

The Notes tab provides an editor for database object notes (extended properties), along with options to save them for distribution.

The Facts tab provides a viewer for database facts/settings (database properties). It also includes information about SQL Server Agent jobs.

DBGizmo is a great utility and it makes a perfect companion to the Database Architecture document. There's even a free edition with many amazing features!

Please visit http://www.DBGizmo.com to learn more.

DBGizmo Tabs