

Ten Common Database Design Mistakes

26 February 2007

by Louis Davidson

No list of mistakes is ever going to be exhaustive. People (myself included) do a lot of really stupid things, at times, in the name of "getting it done." This list simply reflects the database design mistakes that are currently on my mind, or in some cases, constantly on my mind.

NOTE:

I have done this topic two times before. If you're interested in hearing the podcast version, visit Greg Low's super-excellent [SQL Down Under](#). I also presented a boiled down, ten-minute version at PASS for the Simple-Talk booth. Originally there were ten, then six, and today back to ten. And these aren't exactly the same ten that I started with; these are ten that stand out to me as of today.

Before I start with the list, let me be honest for a minute. I used to have a preacher who made sure to tell us before some sermons that he was preaching to himself as much as he was to the congregation. When I speak, or when I write an article, I have to listen to that tiny little voice in my head that helps filter out my own bad habits, to make sure that I am teaching only the best practices. Hopefully, after reading this article, the little voice in your head will talk to you when you start to stray from what is right in terms of database design practices.

So, the list:

1. Poor design/planning
2. Ignoring normalization
3. Poor naming standards
4. Lack of documentation
5. One table to hold all domain values
6. Using identity/guid columns as your only key
7. Not using SQL facilities to protect data integrity
8. Not using stored procedures to access data
9. Trying to build generic objects

10. Lack of testing

Poor design/planning

"If you don't know where you are going, any road will take you there" – George Harrison

Prophetic words for all parts of life and a description of the type of issues that plague many projects these days.

Let me ask you: would you hire a contractor to build a house and then demand that they start pouring a foundation the very next day? Even worse, would you demand that it be done without blueprints or house plans? Hopefully, you answered "no" to both of these. A design is needed make sure that the house you *want* gets built, and that the land you are building it on will not sink into some underground cavern. If you answered yes, I am not sure if anything I can say will help you.

Like a house, a good database is built with forethought, and with proper care and attention given to the needs of the data that will inhabit it; it cannot be tossed together in some sort of reverse implosion.

Since the database is the cornerstone of pretty much every business project, if you don't take the time to map out the needs of the project and how the database is going to meet them, then the chances are that the whole project will veer off course and lose direction. Furthermore, if you don't take the time at the start to get the database design right, then you'll find that any substantial changes in the database structures that you need to make further down the line could have a huge impact on the whole project, and greatly increase the likelihood of the project timeline slipping.

Far too often, a proper planning phase is ignored in favor of just "getting it done". The project heads off in a certain direction and when problems inevitably arise – due to the lack of proper designing and planning – there is "no time" to go back and fix them properly, using proper techniques. That's when the "hacking" starts, with the veiled promise to go back and fix things later, something that happens very rarely indeed.

Admittedly it is impossible to predict every need that your design will have to fulfill and every issue that is likely to arise, but it is important to mitigate against potential problems as much as possible, by careful planning.

Ignoring Normalization

Normalization defines a set of methods to break down tables to their constituent parts until each table represents one and only one "thing", and its columns serve to fully describe only the one "thing" that the table represents.

The concept of normalization has been around for 30 years and is the basis on which SQL and relational databases are implemented. In other words, SQL was created to work with normalized data structures. Normalization is **not** just some plot by database programmers to annoy application programmers (that is merely a satisfying side effect!)

SQL is very additive in nature in that, if you have bits and pieces of data, it is easy to build up a set of values or results. In the **FROM** clause, you take a set of data (a table) and add (JOIN) it to another table. You can add as many sets of data together as you like, to produce the final set you need.

This additive nature is extremely important, not only for ease of development, but also for performance. Indexes are most effective when they can work with the entire key value. Whenever you have to use **SUBSTRING**, **CHARINDEX**, **LIKE**, and so on, to parse out a value that is combined with other values in a single column (for example, to split the last name of a person out of a full name column) the SQL paradigm starts to break down and data becomes become less and less searchable.

So normalizing your data is essential to good performance, and ease of development, but the question always comes up: "How normalized is normalized *enough*?" If you have read any books about normalization, then you will have heard many times that 3rd Normal Form is essential, but 4th and 5th Normal Forms are really useful and, once you get a handle on them, quite easy to follow and well worth the time required to implement them.

In reality, however, it is quite common that not even the first Normal Form is implemented correctly.

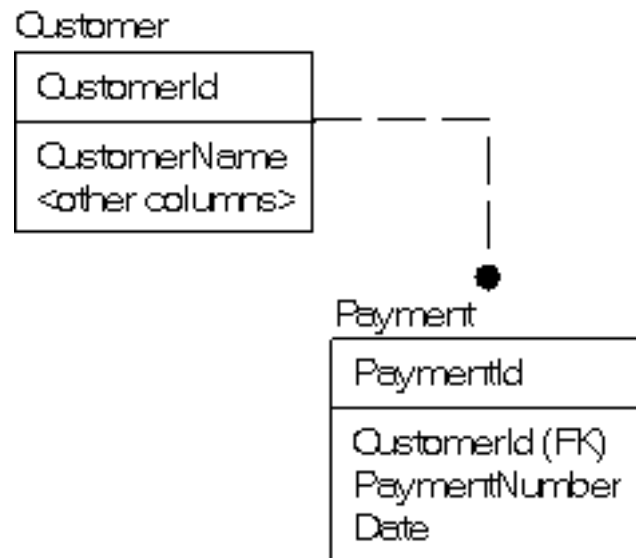
Whenever I see a table with repeating column names appended with numbers, I cringe in horror. And I cringe in horror quite often. Consider the following example **Customer** table:

Customer

CustomerId
CustomerName <other columns>
Payment1
Payment2
Payment3
Payment4
Payment5
Payment6
Payment7
Payment8
Payment9
Payment10
Payment11
Payment12

Are there always 12 payments? Is the order of payments significant? Does a NULL value for a payment mean UNKNOWN (not filled in yet), or a missed payment? And when was the payment made?!?

A payment does not describe a **Customer** and should not be stored in the **Customer** table. Details of payments should be stored in a **Payment** table, in which you could also record extra information about the payment, like when the payment was made, and what the payment was for:



In this second design, each column stores a single unit of information about a single "thing" (a payment), and each row represents a specific instance of a payment.

This second design is going to require a bit more code early in the process but, it is far more likely that you will be able to figure out what is going on in the system without having to hunt down the original programmer and kick their butt...sorry... figure out what they were thinking

Poor naming standards

"That which we call a rose, by any other name would smell as sweet"

This quote from *Romeo and Juliet* by William Shakespeare sounds nice, and it is true from one angle. If everyone agreed that, from now on, a rose was going to be called dung, then we could get over it and it would smell just as sweet. The problem is that if, when building a database for a florist, the designer calls it dung and the client calls it a rose, then you are going to have some meetings that sound far more like an Abbott and Costello routine than a serious conversation about storing information about horticulture products.

Names, while a personal choice, are the first and most important line of documentation for your

application. I will not get into all of the details of how best to name things here— it is a large and messy topic. What I want to stress in this article is the need for **consistency**. The names you choose are not just to enable you to identify the purpose of an object, but to allow all future programmers, users, and so on to quickly and easily understand how a component part of your database was intended to be used, and what data it stores. No future user of your design should need to wade through a 500 page document to determine the meaning of some wacky name.

Consider, for example, a column named, **X304_DSCR**. What the heck does that mean? You might decide, after some head scratching, that it means "X304 description". Possibly it does, but maybe **DSCR** means discriminator, or discretizator?

Unless you have established **DSCR** as a corporate standard abbreviation for description, then **X304_DESCRIPTION** is a much better name, and one leaves nothing to the imagination.

That just leaves you to figure out what the **X304** part of the name means. On first inspection, to me, X304 sounds like more like it should be data in a column rather than a column name. If I subsequently found that, in the organization, there was also an X305 and X306 then I would flag that as an issue with the database design. For maximum flexibility, data is stored in columns, not in column names.

Along these same lines, resist the temptation to include "metadata" in an object's name. A name such as **tblCustomer** or **colVarcharAddress** might seem useful from a development perspective, but to the end user it is just confusing. As a developer, you should rely on being able to determine that a table name is a table name by context in the code or tool, and present to the users clear, simple, descriptive names, such as **Customer** and **Address**.

A practice I strongly advise against is the use of spaces and quoted identifiers in object names. You should avoid column names such as "Part Number" or, in Microsoft style, [Part Number], therefore requiring you users to include these spaces and identifiers in their code. It is annoying and simply unnecessary.

Acceptable alternatives would be **part_number**, **partNumber** or **PartNumber**. Again, consistency is key. If you choose **PartNumber** then that's fine – as long as the column containing invoice numbers is called **InvoiceNumber**, and not one of the other possible variations.

Lack of documentation

I hinted in the intro that, in some cases, I am writing for myself as much as you. This is the topic where that is most true. By carefully naming your objects, columns, and so on, you can make it clear to anyone what it is that your database is modeling. However, this is only step

one in the documentation battle. The unfortunate reality is, though, that "step one" is all too often the *only* step.

Not only will a well-designed data model adhere to a solid naming standard, it will also contain definitions on its tables, columns, relationships, and even default and check constraints, so that it is clear to everyone how they are intended to be used. In many cases, you may want to include sample values, where the need arose for the object, and anything else that you may want to know in a year or two when "future you" has to go back and make changes to the code.

NOTE:

Where this documentation is stored is largely a matter of corporate standards and/or convenience to the developer and end users. It could be stored in the database itself, using extended properties. Alternatively, it might be maintained in the data modeling tools. It could even be in a separate data store, such as Excel or another relational database. My company maintains a metadata repository database, which we developed in order to present this data to end users in a searchable, linkable format. Format and usability is important, but the primary battle is to have the information available and up to date.

Your goal should be to provide enough information that when you turn the database over to a support programmer, they can figure out your minor bugs and fix them (yes, we all make bugs in our code!). I know there is an old joke that poorly documented code is a synonym for "job security." While there is a hint of truth to this, it is also a way to be hated by your coworkers and never get a raise. And no good programmer I know of wants to go back and rework their own code years later. It is best if the bugs in the code can be managed by a junior support programmer while you create the next new thing. Job security along with raises is achieved by being the go-to person for new challenges.

One table to hold all domain values

"One Ring to rule them all and in the darkness bind them"

This is all well and good for fantasy lore, but it's not so good when applied to database design, in the form of a "ruling" domain table. Relational databases are based on the fundamental idea that every object represents one and only one thing. There should never be any doubt as to what a piece of data refers to. By tracing through the relationships, from column name, to table name, to primary key, it should be easy to examine the relationships and know exactly what a piece of data means.

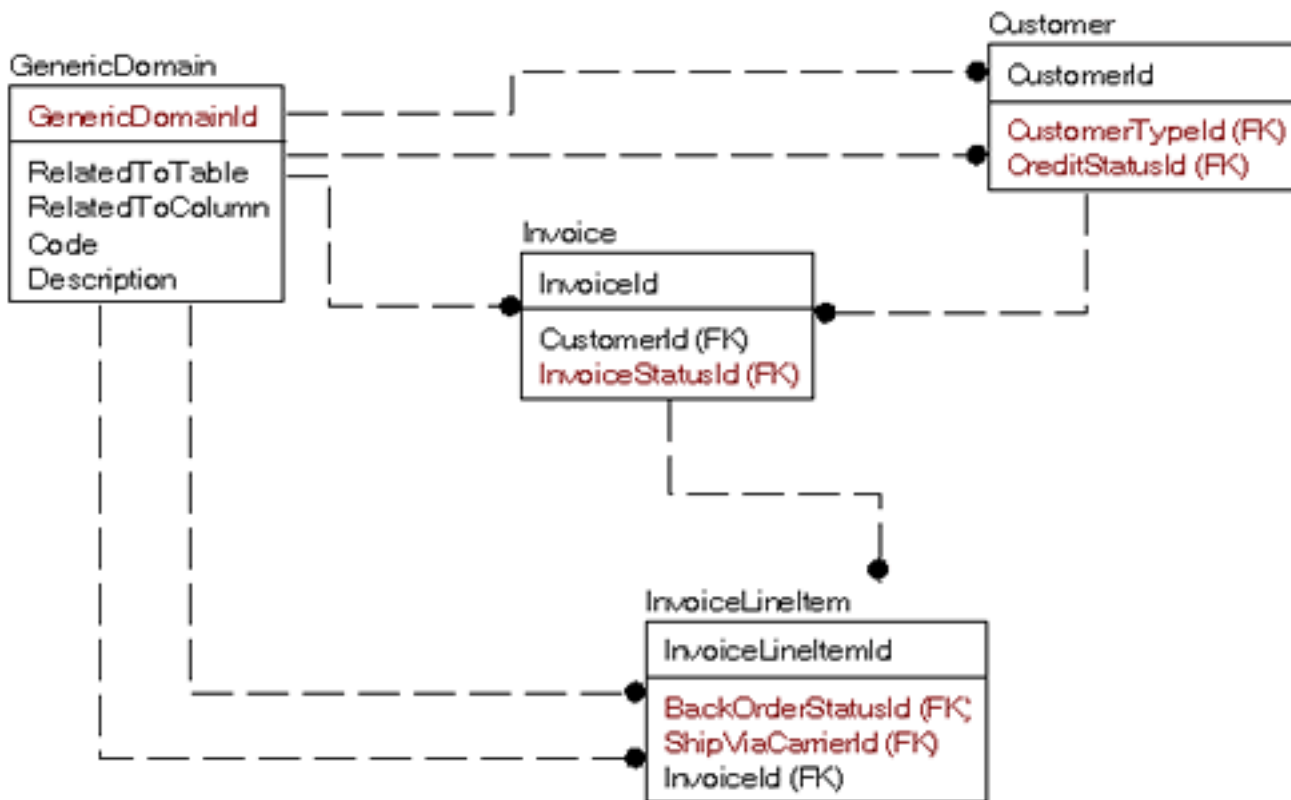
The big myth perpetrated by architects who don't really understand relational database architecture (me included early in my career) is that the more tables there are, the more complex the design will be. So, conversely, shouldn't condensing multiple tables into a single

"catch-all" table simplify the design? It does sound like a good idea, but at one time giving Pauly Shore the lead in a movie sounded like a good idea too.

For example, consider the following model snippet where I needed domain values for:

- Customer CreditStatus
- Customer Type
- Invoice Status
- Invoice Line Item BackOrder Status
- Invoice Line Item Ship Via Carrier

On the face of it that would be five domain tables...but why not just use one generic domain table, like this?



This may seem a very clean and natural way to design a table for all but the problem is that it is just not very natural to work with in SQL. Say we just want the domain values for the **Customer** table:

```

SELECT *
FROM Customer
  JOIN GenericDomain as CustomerType
    ON Customer.CustomerTypeId = CustomerType.GenericDomainId
     and CustomerType.RelatedToTable = 'Customer'
    
```

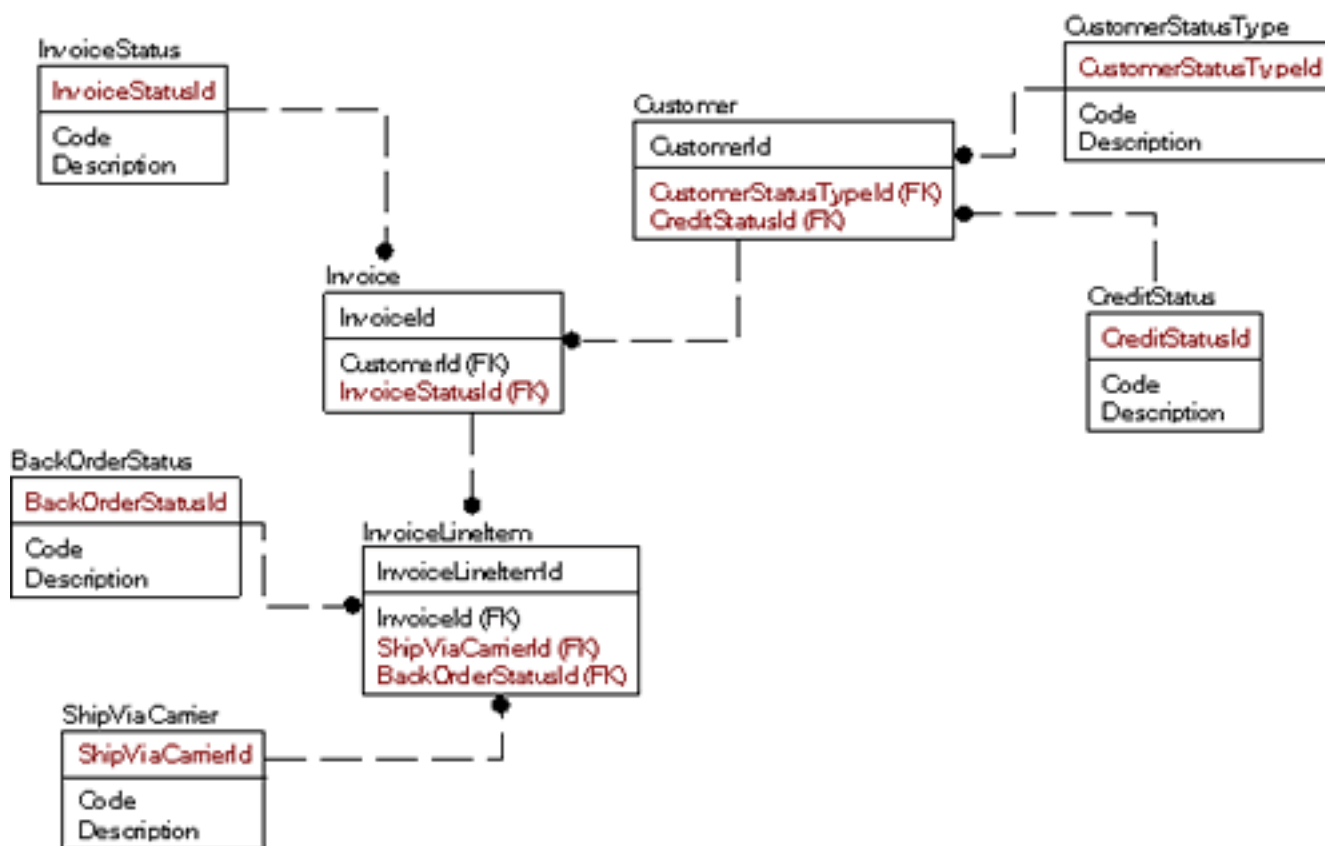
```

and CustomerType.RelatedToColumn = 'CustomerId'
JOIN GenericDomain as CreditStatus
ON Customer.CreditStatusId = CreditStatus.GenericDomainId
and CreditStatus.RelatedToTable = 'Customer'
and CreditStatus.RelatedToColumn = ' CreditStatusId'

```

As you can see, this is far from being a natural join. It comes down to the problem of mixing apples with oranges. At first glance, domain tables are just an abstract concept of a container that holds text. And from an implementation centric standpoint, this is quite true, but it is not the correct way to build a database. In a database, the process of normalization, as a means of breaking down and isolating data, takes every table to the point where one row represents one thing. And each domain of values is a distinctly different thing from all of the other domains (unless it is not, in which case the one table will suffice.). So what you do, in essence, is normalize the data on each usage, spreading the work out over time, rather than doing the task once and getting it over with.

So instead of the single table for all domains, you might model it as:



Looks harder to do, right? Well, it is initially. Frankly it took me longer to flesh out the example tables. But, there are quite a few tremendous gains to be had:

- Using the data in a query is much easier:

```

SELECT *
FROM Customer
  JOIN CustomerType
    ON Customer.CustomerTypeId = CustomerType.CustomerTypeId
  JOIN CreditStatus
    ON Customer.CreditStatusId = CreditStatus.CreditStatusId

```

- Data can be validated using foreign key constraints very naturally, something not feasible for the other solution unless you implement ranges of keys for every table – a terrible mess to maintain.
- If it turns out that you need to keep more information about a **ShipViaCarrier** than just the code, 'UPS', and description, 'United Parcel Service', then it is as simple as adding a column or two. You could even expand the table to be a full blown representation of the businesses that are carriers for the item.
- All of the smaller domain tables will fit on a single page of disk. This ensures a single read (and likely a single page in cache). If the other case, you might have your domain table spread across many pages, unless you cluster on the referring table name, which then could cause it to be more costly to use a non-clustered index if you have many values.
- You can still have one editor for all rows, as most domain tables will likely have the same base structure/usage. And while you would lose the ability to query all domain values in one query easily, why would you want to? (A union query could easily be created of the tables easily if needed, but this would seem an unlikely need.)

I should probably rebut the thought that might be in your mind. "What if I need to add a new column to all domain tables?" For example, you forgot that the customer wants to be able to do custom sorting on domain values and didn't put anything in the tables to allow this. This is a fair question, especially if you have 1000 of these tables in a very large database. First, this rarely happens, and when it does it is going to be a major change to your database in either way.

Second, even if this became a task that was required, SQL has a complete set of commands that you can use to add columns to tables, and using the system tables it is a pretty straightforward task to build a script to add the same column to hundreds of tables all at once. That will not be as easy of a change, but it will not be so much more difficult to outweigh the large benefits.

The point of this tip is simply that it is better to do the work upfront, making structures solid and maintainable, rather than trying to attempt to do the least amount of work to start out a project. By keeping tables down to representing one "thing" it means that most changes will only affect one table, after which it follows that there will be less rework for you down the road.

Using identity/guid columns as your only key

First Normal Form dictates that all rows in a table must be uniquely identifiable. Hence, every table should have a primary key. SQL Server allows you to define a numeric column as an **IDENTITY** column, and then automatically generates a unique value for each row.

Alternatively, you can use **NEWID()** (or **NEWSEQUENTIALID()**) to generate a random, 16 byte unique value for each row. These types of values, when used as keys, are what are known as **surrogate keys**. The word surrogate means "something that substitutes for" and in this case, a surrogate key should be the stand-in for a natural key.

The problem is that too many designers use a surrogate key column as the *only* key column on a given table. The surrogate key values have no actual meaning in the real world; they are just there to uniquely identify each row.

Now, consider the following **Part** table, whereby **PartID** is an **IDENTITY** column and is the primary key for the table:

PartID	PartNumber	Description
1	XXXXXXXXX	The X part
2	XXXXXXXXX	The X part
3	YYYYYYYYY	The Y part

How many rows are there in this table? Well, there seem to be three, but are rows with **PartIDs** 1 and 2 actually the same row, duplicated? Or are they two different rows that should be unique but were keyed in incorrectly?

The rule of thumb I use is simple. If a human being could not pick which row they want from a table without knowledge of the surrogate key, then you need to reconsider your design. This is why there should be a key of some sort on the table to guarantee uniqueness, in this case likely on **PartNumber**.

In summary: as a rule, each of your tables should have a natural key that means something to the user, and can uniquely identify each row in your table. In the very rare event that you cannot find a natural key (perhaps, for example, a table that provides a log of events), then use an artificial/surrogate key.

Not using SQL facilities to protect data integrity

All fundamental, non-changing business rules should be implemented by the relational engine. The **base rules** of nullability, string length, assignment of foreign keys, and so on, should all

be defined **in the database**.

There are many different ways to import data into SQL Server. If your base rules are defined in the database itself can you guarantee that they will never be bypassed and you can write your queries without ever having to worry whether the data you're viewing adheres to the base business rules.

Rules that are optional, on the other hand, are wonderful candidates to go into a business layer of the application. For example, consider a rule such as this: "For the first part of the month, no part can be sold at more than a 20% discount, without a manager's approval".

Taken as a whole, this rule smacks of being rather messy, not very well controlled, and subject to frequent change. For example, what happens when next week the maximum discount is 30%? Or when the definition of "first part of the month" changes from 15 days to 20 days? Most likely you won't want go through the difficulty of implementing these complex temporal business rules in SQL Server code – the business layer is a great place to implement rules like this.

However, consider the rule a little more closely. There are elements of it that will probably never change. E.g.

- The maximum discount it is ever possible to offer
- The fact that the approver must be a manager

These aspects of the business rule very much ought to get enforced by the database and design. Even if the substance of the rule is implemented in the business layer, you are still going to have a table in the database that records the size of the discount, the date it was offered, the ID of the person who approved it, and so on. On the **Discount** column, you should have a **CHECK** constraint that restricts the values allowed in this column to between 0.00 and 0.90 (or whatever the maximum is). Not only will this implement your "maximum discount" rule, but will also guard against a user entering a 200% or a negative discount by mistake. On the **ManagerID** column, you should place a foreign key constraint, which reference the Managers table and ensures that the ID entered is that of a real manager (or, alternatively, a trigger that selects only **Employeeids** corresponding to managers).

Now, at the very least we can be sure that the data meets the very basic rules that the data must follow, so we never have to code something like this in order to check that the data is good:

```
SELECT CASE WHEN discount < 0 then 0 else WHEN discount > 1 then 1...
```

We can feel safe that data meets the basic criteria, every time.

Not using stored procedures to access data

Stored procedures are your friend. Use them whenever possible as a method to insulate the database layer from the users of the data. Do they take a bit more effort? Sure, initially, but what good thing doesn't take a bit more time? Stored procedures make database development much cleaner, and encourage collaborative development between your database and functional programmers. A few of the other interesting reasons that stored procedures are important include the following.

Maintainability

Stored procedures provide a known interface to the data, and to me, this is probably the largest draw. When code that accesses the database is compiled into a different layer, performance tweaks cannot be made without a functional programmer's involvement. Stored procedures give the database professional the power to change characteristics of the database code without additional resource involvement, making small changes, or large upgrades (for example changes to SQL syntax) easier to do.

Encapsulation

Stored procedures allow you to "encapsulate" any structural changes that you need to make to the database so that the knock on effect on user interfaces is minimized. For example, say you originally modeled one phone number, but now want an unlimited number of phone numbers. You could leave the single phone number in the procedure call, but store it in a different table as a stopgap measure, or even permanently if you have a "primary" number of some sort that you always want to display. Then a stored proc could be built to handle the other phone numbers. In this manner the impact to the user interfaces could be quite small, while the code of stored procedures might change greatly.

Security

Stored procedures can provide specific and granular access to the system. For example, you may have 10 stored procedures that all update table X in some way. If a user needs to be able to update a particular column in a table and you want to make sure they never update any others, then you can simply grant to that user the permission to execute just the one procedure out of the ten that allows them perform the required update.

Performance

There are a couple of reasons that I believe stored procedures enhance performance. First, if a newbie writes ratty code (like using a cursor to go row by row through an entire ten million

row table to find one value, instead of using a WHERE clause), the procedure can be rewritten without impact to the system (other than giving back valuable resources.) The second reason is plan reuse. Unless you are using dynamic SQL calls in your procedure, SQL Server can store a plan and not need to compile it every time it is executed. It's true that in every version of SQL Server since 7.0 this has become less and less significant, as SQL Server gets better at storing plans ad hoc SQL calls (see note below). However, stored procedures still make it easier for plan reuse and performance tweaks. In the case where ad hoc SQL would actually be faster, this can be coded into the stored procedure seamlessly.

In 2005, there is a database setting (**PARAMETERIZATION FORCED**) that, when enabled, will cause all queries to have their plans saved. This does not cover more complicated situations that procedures would cover, but can be a big help. There is also a feature known as **plan guides**, which allow you to override the plan for a known query type. Both of these features are there to help out when stored procedures are not used, but stored procedures do the job with no tricks.

And this list could go on and on. There are drawbacks too, because nothing is ever perfect. It can take longer to code stored procedures than it does to just use ad hoc calls. However, the amount of time to design your interface and implement it is well worth it, when all is said and done.

Trying to code generic T-SQL objects

I touched on this subject earlier in the discussion of generic domain tables, but the problem is more prevalent than that. Every new T-SQL programmer, when they first start coding stored procedures, starts to think "I wish I could just pass a table name as a parameter to a procedure." It does sound quite attractive: one generic stored procedure that can perform its operations on any table you choose. However, this should be avoided as it can be very detrimental to performance and will actually make life more difficult in the long run.

T-SQL objects do not do "generic" easily, largely because lots of design considerations in SQL Server have clearly been made to facilitate reuse of plans, not code. SQL Server works best when you minimize the unknowns so it can produce the best plan possible. The more it has to generalize the plan, the less it can optimize that plan.

Note that I am not specifically talking about dynamic SQL procedures. Dynamic SQL is a great tool to use when you have procedures that are not optimizable / manageable otherwise. A good example is a search procedure with many different choices. A precompiled solution with multiple OR conditions might have to take a worst case scenario approach to the plan and yield weak results, especially if parameter usage is sporadic.

However, the main point of this tip is that you should avoid coding very generic objects, such

as ones that take a table name and twenty column names/value pairs as a parameter and lets you update the values in the table. For example, you could write a procedure that started out:

```
CREATE PROCEDURE updateAnyTable
@tableName sysname,
@columnName1 sysname,
@columnName1Value varchar(max)
@columnName2 sysname,
@columnName2Value varchar(max)
...
```

The idea would be to dynamically specify the name of a column and the value to pass to a SQL statement. This solution is no better than simply using ad hoc calls with an UPDATE statement. Instead, when building stored procedures, you should build specific, dedicated stored procedures for each task performed on a table (or multiple tables.) This gives you several benefits:

- Properly compiled stored procedures can have a single compiled plan attached to it and reused.
- Properly compiled stored procedures are more secure than ad-hoc SQL or even dynamic SQL procedures, reducing the surface area for an injection attack greatly because the only parameters to queries are search arguments or output values.
- Testing and maintenance of compiled stored procedures is far easier to do since you generally have only to search arguments, not that tables/columns/etc exist and handling the case where they do not

A nice technique is to build a code generation tool in your favorite programming language (even T-SQL) using SQL metadata to build very specific stored procedures for every table in your system. Generate all of the boring, straightforward objects, including all of the tedious code to perform error handling that is so essential, but painful to write more than once or twice.

In my Apress book, [Pro SQL Server 2005 Database Design and Optimization](#), I provide several such "templates" (manly for triggers, abut also stored procedures) that have all of the error handling built in, I would suggest you consider building your own (possibly based on mine) to use when you need to manually build a trigger/procedure or whatever.

Lack of testing

When the dial in your car says that your engine is overheating, what is the first thing you blame? The engine. Why don't you immediately assume that the dial is broken? Or something else minor? Two reasons:

- The engine is the most important component of the car and it is common to blame the

- most important part of the system first.
- It is all too often true.

As database professionals know, the first thing to get blamed when a business system is running slow is the database. Why? First because it is the central piece of most any business system, and second because it also is all too often true.

We can play our part in dispelling this notion, by gaining deep knowledge of the system we have created and understanding its limits through **testing**.

But let's face it; testing is the first thing to go in a project plan when time slips a bit. And what suffers the most from the lack of testing? Functionality? Maybe a little, but users will notice and complain if the "Save" button doesn't actually work and they cannot save changes to a row they spent 10 minutes editing. What really gets the shaft in this whole process is deep system testing to make sure that the design you (presumably) worked so hard on at the beginning of the project is actually implemented correctly.

But, you say, the users accepted the system as working, so isn't that good enough? The problem with this statement is that what user acceptance "testing" usually amounts to is the users poking around, trying out the functionality that they understand and giving you the thumbs up if their little bit of the system works. Is this reasonable testing? Not in any other industry would this be vaguely acceptable. Do you want your automobile tested like this? "Well, we drove it slowly around the block once, one sunny afternoon with no problems; it is good!" When that car subsequently "failed" on the first drive along a freeway, or during the first drive through rain or snow, then the driver would have every right to be very upset.

Too many database systems get tested like that car, with just a bit of poking around to see if individual queries and modules work. The first real test is in production, when users attempt to do real work. This is especially true when it is implemented for a single client (even worse when it is a corporate project, with management pushing for completion more than quality).

Initially, major bugs come in thick and fast, especially performance related ones. If the first time you have tried a full production set of users, background process, workflow processes, system maintenance routines, ETL, etc, is on your system launch day, you are extremely likely to discover that you have not anticipated all of the locking issues that might be caused by users creating data while others are reading it, or hardware issues cause by poorly set up hardware. It can take weeks to live down the cries of "SQL Server can't handle it" even after you have done the proper tuning.

Once the major bugs are squashed, the fringe cases (which are pretty rare cases, like a user entering a negative amount for hours worked) start to raise their ugly heads. What you end up with at this point is software that irregularly fails in what seem like weird places (since large

quantities of fringe bugs will show up in ways that aren't very obvious and are really hard to find.)

Now, it is far harder to diagnose and correct because now you have to deal with the fact that users are working with live data and trying to get work done. Plus you probably have a manager or two sitting on your back saying things like "when will it be done?" every 30 seconds, even though it can take days and weeks to discover the kinds of bugs that result in minor (yet important) data aberrations. Had proper testing been done, it would never have taken weeks of testing to find these bugs, because a proper test plan takes into consideration all possible types of failures, codes them into an automated test, and tries them over and over. Good testing won't find all of the bugs, but it will get you to the point where most of the issues that correspond to the original design are ironed out.

If everyone insisted on a strict testing plan as an integral and immutable part of the database development process, then maybe someday the database won't be the first thing to be fingered when there is a system slowdown.

Summary

Database design and implementation is the cornerstone of any data centric project (read 99.9% of business applications) and should be treated as such when you are developing. This article, while probably a bit preachy, is as much a reminder to me as it is to anyone else who reads it. Some of the tips, like planning properly, using proper normalization, using a strong naming standards and documenting your work– these are things that even the best DBAs and data architects have to fight to make happen. In the heat of battle, when your manager's manager's manager is being berated for things taking too long to get started, it is not easy to push back and remind them that they pay you now, or they pay you later. These tasks pay dividends that are very difficult to quantify, because to quantify success you must fail first. And even when you succeed in one area, all too often other minor failures crop up in other parts of the project so that some of your successes don't even get noticed.

The tips covered here are ones that I have picked up over the years that have turned me from being mediocre to a good data architect/database programmer. None of them take extraordinary amounts of time (except perhaps design and planning) but they all take more time upfront than doing it the "easy way". Let's face it, if the easy way were that easy in the long run, I for one would abandon the harder way in a second. It is not until you see the end result that you realize that success comes from starting off right as much as finishing right.

© Simple-Talk.com