Defusing Database Time Bombs: Avoiding the Need to Refactor Databases

27 February 2015

*by* [*Thomas LeBlanc*](https://www.simple-talk.com/author/thomas-leblanc/)

*FROM:* [*https://www.simple-talk.com/sql/t-sql-programming/defusing-database-time-bombs-avoiding-the-need-to-refactor-databases/?utm\_source=ssc&utm\_medium=publink&utm\_content=defusingdbs*](https://www.simple-talk.com/sql/t-sql-programming/defusing-database-time-bombs-avoiding-the-need-to-refactor-databases/?utm_source=ssc&utm_medium=publink&utm_content=defusingdbs)

Where applications are evolved by gradually molding them to a growing understanding of the business domain, this presents great challenges to database development. If databases are designed too loosely, and initial errors are allowed to fester, the results become harder and harder to refactor until eventually they constitute a database time bomb. Thomas LeBlanc describes how to avoid a few basic, but very common, database time bombs.

Poor database design choices, which subsequently have to be refactored, reveal themselves in many different ways. Over the years I've seen everything from unfeasibly wide tables that are incredibly expensive to query, to inconsistently or incomprehensibly-named objects that are impossible to maintain, to poor data types choices that lead to wasted storage space, contorted data logic, and expensive data type conversions, to security mistakes and malpractices that threaten the very integrity of the data.

The vast majority of the problems I see in database designs are actually fairly basic and relate to fundamental problems with the requirements gathering and database normalization techniques. These can and will cause big problems and time-consuming refactoring efforts further down the line. In this article, I'll review briefly some practices that can help prevent these sorts of database '[code smells](https://www.simple-talk.com/books/sql-books/119-sql-code-smells/?utm_source=simpletalk&utm_medium=weblink&utm_content=defusingdbs)' from creeping into our systems, and becoming database time-bombs. Then, I'll take a more detailed look at what I consider five of the most common data design flaws that eventually lead to the need to perform database refactoring, for reasons of data integrity, code maintenance, query performance, or all three:

1. Identity column as Primary Key with no natural candidate key
2. Combining multiple values in one column
3. Not using Foreign Keys to enforce referential integrity
4. Data stored in the wrong data type – for example, a numeric data type that we then need to change to a textual data type.
5. Poorly named or sized columns, which leads to:
   1. Changing the size of a column
   2. Changing a column name

As we progress, I'll discuss some of the challenges inherent in the refactoring efforts required to remove these problems, and provide references for further details of the practicalities of performing database refactoring while minimizing application-downtime and user-disruption.

**Practices to Help Minimize Database Refactoring**

Database 'code smells' can creep into our systems at many different stages. Often the problems start right at the very beginning, with incomplete requirements gathering, leading to incorrect design choices. A second common early entry point for code smells, and the most common in my experience, is neglect of database normalization best practices, which leads to all sort of problems, such as 'repeating groups' and non-atomic values in columns.

Even if we start from a solid database design, based on well-researched and understood requirements, design flaws can and do creep in over time, as requirements change, or changes are enforced due to company mergers and reorganizations and [changes in legislation](https://www.simple-talk.com/opinion/opinion-pieces/it-compliance-and-software-development/?utm_source=simpletalk&utm_medium=weblink&utm_content=defusingdbs). This problem is far worse in organizations where there are no documented 'operating procedures' offering consistent guidelines for creating or modifying tables and their related objects. The content of these "SOP" documents will vary by organization, but broadly they will describe what data, in business terms, each table and column should store, naming conventions, data types and sizes, and so on.

**Involve a Data Architect/DBA in Requirement Gathering**

All database designs must start from a solid understanding of the business requirements of the application, the type and volume of data that needs to be stored, a shared understating about what each data element really means, in business terms, and how the data will be used by other applications, and for reporting and so on.

Some developers are capable of producing a competent database design, although many more, at least in my experience, are capable only of letting a tool generate a database 'design' that matches their lovingly-crafted GUI. The best way to avoid some serious design pitfall and subsequent refactoring, especially in the latter case, is for the development team to include the DBA or database developer in their application requirement gathering meetings. They could even go a step further and have them validate the functional specification or technical requirements before even starting work on the design phase.

All of this understanding should be poured into the conceptual view of the database design. I create my Data Diagrams in Visio, but failing that even taking time to draw it out on a piece of paper is highly beneficial in terms of spotting a relationship or table or column I've missed, and saves many, many hours of time later, trying to fix a poor design.

Finally, don't forget that you need requirements gathering and up-front design for reporting as well as transaction processing. Designing the database without understanding what's required for reporting is a common mistake I have seen in the last 10 years. The developers are not responsible for the reporting, just the user interface, so there is no planning during their requirements gathering to take into account the reports.

I've also seen cases where developers 'simplify' a data architect's design during the build, not realizing that the design was that way specifically to incorporate reporting requirements. A classic example is an initial design that establishes a direct relationship to tell us easily that "this row was added to table B because a user performed action A". The developers then remove this direct relationship to simplify the design, but in the process make cause-and-effect reporting much harder.

**Document a Standard Operating Procedure**

Before starting work on the physical design, it's worth enshrining in an "SOP" document some guidelines that will direct certain fundamental aspects of the database design, and how it will be verified, as well as some naming standards.

The naming standards provide a checklist of [naming conventions, allowable abbreviation, data types and sizes](https://www.simple-talk.com/sql/t-sql-programming/laying-out-sql-code/?utm_source=simpletalk&utm_medium=weblink&utm_content=defusingdbs) and so on. I've always found this very useful during the design phase, to validate that I'm on the right track. Following are examples of naming conventions I have seen in a SOP, but are not inclusive of all topics required:

* **Naming Conventions** – column names should be as descriptive as possible without being overly long. For example, ProductNumber rather than Number or No. (for Product table)
* **Abbreviations** – a list of abbreviations used in column names
  + **ID** – Identifier
  + **SSN** – Social Security Number
* **Data Types** –
  + Use int for all integer types (or list integer types and when to use them)
  + Use varchar and not char (unless there is a case for char, such as a StateAbbreviation column, which is always 2 characters)
  + Money for amount type columns and decimal for ratios or percentages
  + Minimum size for descriptive columns – 40 characters

Beyond naming conventions, an SOP for a database design will cover operational aspects such as, for example, the need for the architect review the database structures before they are created as code, and how database testing will be done. It will also guide certain fundamental aspects of how the database should be designed and accessed covering, for example:

* **Access methods** – such as via stored procedures
* **Security implementation** – roles, permissions and so on
* **Standard domains** – for example, "*a person's age will be stored as a* tinyint*, with a range of 0-140*", or "*a person's first name will be 35 characters, and may not be an empty string*".

**Follow Basic Database Normalization Rules**

This is far too big a topic to attempt to cover in any detail here. However, most of the specific design problems I'll cover later in this article are caused by a failure to implement [basic Database Normalization rules](https://www.simple-talk.com/sql/t-sql-programming/database-normalization-basics/?utm_source=simpletalk&utm_medium=weblink&utm_content=defusingdbs), at least to Third Normal Form. For certain OLTP (Online Transaction Processing) database, use of Fourth and Fifth Normal Forms could also be required.

The basic aim on normalization, as stated in most articles on the topic, is to eradicate problems such as columns containing compound values, or redundant data (the same value stored over and again), by applying a progressive set of rules, according to "Normal Forms':

* **First Normal Form** – all columns should relate to the key of the table
* **Second Normal Form** – the table has multiple key columns to identify other columns
* **Third Normal Form** – columns in the table should be descriptive of the Key, not a non-key

However, put simply, the most important goal of normalization is to define precisely the **cardinality** between each table. Understanding the exact relationship between one piece of data and another, in turn, requires a solid understanding of the business requirements. If your business rules limit each individual customer to making only one order of one product, to be sent to one address and shipped immediately, then you may only need one table. If this sounds nothing like what you want for your business, then normalization is important.

**Refactoring Databases: the practicalities**

Changing the database schema can sometimes be a tricky process in a poorly-designed database, and one that requires a lot of care and planning, both in terms of how best to implement the required change, and how to ensure that all objects affected by the change are located and modified appropriately.

Next, we need to ensure that the whole refactoring process causes minimum downtime and user disruption. This is a big topic in itself, and the focus of this article is on how to avoid the need for refactoring in the first place, so we won't be going into details. However, as we proceed through the examples, I will review some of the common tools and tactics we can use to try to make sure any database refactoring proceeds as smoothly as possible, and provide references to articles that cover these topics in more depth.

Over the coming sections, I'll walk through examples of the five most common database time bombs that I encounter, and how we can refactor a database to remove them. The code download file for this article (see the **Downloads** link in the box to the right of the article title) contains the code to create a DBRefactor database (Script 1; not shown), and then various scripts to create and populate the tables used in the article (I also provide backups of the database I created). I populate the tables from existing tables in the AdventureWorks2012 database.

**Identity Column as Primary Key with no Natural Candidate Key**

This is probably the single most common database design mistake that I see, usually the sign of a database designed by someone with no practical knowledge of database normalization. The designer heard that an IDENTITY column is faster for joins, so he or she decides to create every table with an IDENTITY column as a primary key. The problem with this design is that since each table is an entity that represents a certain type of business data (such as a Product), and each row represent a different instance of that entity, so every table also needs a natural key, what I often call the business key, which a human rather than just SQL Server could use to identify each row in a table.

Script 2, reproduced below, creates a Product table with ProductID, an IDENTITY column as the primary key (the code download also contains a script to populate this table).

CREATE TABLE dbo.Product

(

ProductID int IDENTITY(1, 1)

NOT NULL ,

Name VARCHAR(50) NOT NULL ,

ProductNumber varchar(25) NOT NULL ,

Color varchar(15) NULL ,

StandardCost money NOT NULL ,

ListPrice money NOT NULL ,

Size varchar(5) NULL ,

SizeUnitMeasureCode char(3) NULL ,

DaysToManufacture int NOT NULL ,

ProductSubcategoryID int NULL ,

CONSTRAINT PK\_Product\_ProductID PRIMARY KEY CLUSTERED

( ProductID ASC )

);

GO

**Script 2**

This table does have a candidate natural key, ProductNumber, but because the designer has not enforced the uniqueness of this column, it means that we can insert what is essentially the same row into this table multiple times. Then, if a stored procedure or application code queries the Product table by ProductNumber, without an ORDERBY statement, the resulting related records can have different ProductID values used on multiple rows in the related table. This means that there will be more than one row for the same product, with catastrophic effects to any downstream accounting systems that do joins via the surrogate identity key.

An easy solution is to enforce the uniqueness of ProductNumber by creating a unique constraint or unique index on this column. Script 3 creates a UNIQUE index (in the code download file, Script 3 inserts some data into the Product table before creating the index).

CREATE UNIQUE NONCLUSTERED INDEX AK\_ProductNumber

ON dbo.Product

( ProductNumber ASC )

GO

**Script 3**

Even though there was no refactoring of code directly related to this change, the use of good database normalization starts with this standard.

Most well-designed lookup tables will, in my experience, have both a surrogate key *and* a natural key; having only the latter can sometimes cause problems too. Consider the alternative case where only the natural key exists on our Product table, and so is the primary key to which all child tables refer, through their foreign keys. However, due to changing requirements, we need to change the data type of the natural key. We are now faced with a very significant refactoring challenge. However, with a surrogate key in place in the form of an integer IDENTITY column, child tables are unaffected by any changes to the natural key, and so the refactoring task is easier.

**Data stored in the wrong data type**

Consider the case where the original requirement for a database design stipulated a numeric data type, such as in int, for certain business key. However, somewhere along the line and for some reason, the requirement for the business key changes; it now needs to be alphanumeric, and so we need to refactor the table to change the column to a character data type. This sort of requirement change, for example in response to business acquisitions or business restructuring, is sometimes unavoidable.

Let's see this in action. First, run Script 4 (not shown but available in code download) to create and populate a table called ProductIntegerAK, which is essentially a copy of the Product table in Script 2, except that the ProductNumber column is a int, rather than varchar(15).

Now, we need to refactor the table to change the data type, as shown in Script 5.

ALTER TABLE dbo.ProductIntegerAK

ALTER COLUMN ProductNumber VARCHAR(25) NOT NULL

GO

**Script 5**

However, this results in the following error because we neglected to first drop and recreate any indexes that use this column.

Msg 5074, Level 16, State 1, Line 1

The index 'AK\_ProductIntegerAK\_ProductNumber' is dependent on column 'ProductNumber'.

Msg 4922, Level 16, State 9, Line 1

ALTER TABLE ALTER COLUMN ProductNumber failed because one or more objects access this column.

Script 6 drops the UNIQUE index, updates the date type of the column, and then adds the UNIQUE Index back to table.

USE DBRefactor

GO

DROP INDEX AK\_ProductIntegerAK\_ProductNumber

ON dbo.ProductIntegerAK

GO

ALTER TABLE dbo.ProductIntegerAK

ALTER COLUMN ProductNumber VARCHAR(25) NOT NULL

CREATE UNIQUE NONCLUSTERED INDEX AK\_ProductIntegerAK\_ProductNumber

ON dbo.ProductIntegerAK

( ProductNumber ASC )

GO

**Script 6**

Of course, this is just the database part of the change. Now, we need to update any stored procedures, user-defined functions, triggers and application code that reference this column.

Script 7 shows an example of stored procedure to return a product and then demonstrates how to refactor the stored procedure to reflect the data type change.

USE DBRefactor;

GO

CREATE PROCEDURE dbo.ProductIntegerAK\_Get\_ProductID

@intProductNumber INT

AS

SELECT ProductID

FROM dbo.ProductIntegerAK

WHERE ProductNumber = @intProductNumber;

GO

ALTER PROCEDURE dbo.ProductIntegerAK\_Get\_ProductID

@txtProductNumber VARCHAR(25)

AS

SELECT ProductID

FROM dbo.ProductIntegerAK

WHERE ProductNumber = @txtProductNumber;

GO

**Script 7**

Note that I'm using Hungarian notation for the variables simply to distinguish them clearly in this example. I wouldn't recommend it generally precisely for the reason that it can lead to more refactoring! Hopefully, your SOP document will provide variable naming standards.

Finding all the dependencies is sometimes the hardest part of a refactoring exercise. Phil Factor's article, [Exploring your database schema with SQL](https://www.simple-talk.com/sql/t-sql-programming/exploring-your-database-schema-with-sql/?utm_source=simpletalk&utm_medium=weblink&utm_content=defusingdbs), provides code to help interrogate the objects dependency chain. However, some things don't show up in the dependency chain such as variables and temporary tables that hold the values from that column, and so you might also need to perform a string search, either using [T-SQL](https://www.simple-talk.com/sql/t-sql-programming/searching-for-strings-in-sql-server-databases-/?utm_source=simpletalk&utm_medium=weblink&utm_content=defusingdbs), or a tool such as [SQL Search](http://www.red-gate.com/products/sql-development/sql-search/?utm_source=simpletalk&utm_medium=publink&utm_content=defusingdbs&__hstc=81186632.3c044b552fdca0c90c6745f4b52b980d.1417613221615.1426103415216.1426249070096.22&__hssc=81186632.2.1426249070096&__hsfp=2502449689).

Think carefully when [choosing data types](http://sqlblog.com/blogs/aaron_bertrand/archive/2009/10/12/bad-habits-to-kick-using-the-wrong-data-type.aspx) but also think about simple measures to mitigate the cost of any possible future refactoring. One example is having a surrogate key as well as natural key, as discussed earlier. Another example would be using a char(10) for a BirthDate column; I have seen this when importing data from text files. Then, code and reports have to format the date column and get 'bad date' errors. This problem would be caught at data entry if had the column used a Date data type.

As a final consideration, remember that this sort of refactoring can impact database level resources like the transaction log. All modifications get logged, so in the above case the change to the table will be logged as well as the drop/create of an index. Also, if Replication or Database Mirroring is active, then you'll also need to replicate or mirror these changes to the secondary server, which takes time and planning.

**Combining Multiple Values in One Column**

Let's assume that, for convenience, an application or report needs to display in a drop-down the combined product code and product name. Of course, with a properly normalized design this is easy to do, using a query or view, to display the data as required in the drop-down. However, instead, the novice designer simply combined into a single column called Product both the business key and the product name along with an indication of size and color. Script 8 shows the structure of the table.

USE DBRefactor;

GO

CREATE TABLE dbo.ProductCombinedColumn

(

ProductID INT IDENTITY(1, 1)

NOT NULL ,

Product VARCHAR(100) NOT NULL ,

Color VARCHAR(15) NULL ,

StandardCost MONEY NOT NULL ,

ListPrice MONEY NOT NULL ,

Size VARCHAR(5) NULL ,

DaysToManufacture INT NOT NULL ,

SubcategoryID INT NULL ,

CONSTRAINT PK\_ProductCombinedColumn\_ProductID PRIMARY KEY CLUSTERED

( ProductID ASC )

);

GO

CREATE UNIQUE NONCLUSTERED INDEX AK\_Product

ON dbo.ProductCombinedColumn(Product ASC);

GO

--Insert data from AdventureWorks2012.Production.Product

-- (see code download file)

-- Deal with NULL values

UPDATE dbo.ProductCombinedColumn

SET SubcategoryID = 38

WHERE SubcategoryID IS NULL;

GO

-- Make SubCategoryID NOT NULL

ALTER TABLE dbo.ProductCombinedColumn

ALTER COLUMN SubcategoryID INT NOT NULL;

GO

-- Add FK to ProductSubCategory

-- First, create the ProductCategory and ProductSubCategory tables

-- See Scripts 9 and 10

ALTER TABLE dbo.ProductCombinedColumn WITH CHECK

ADD CONSTRAINT FK\_ProductCombinedColumn\_ProductSubcategory\_SubcategoryID

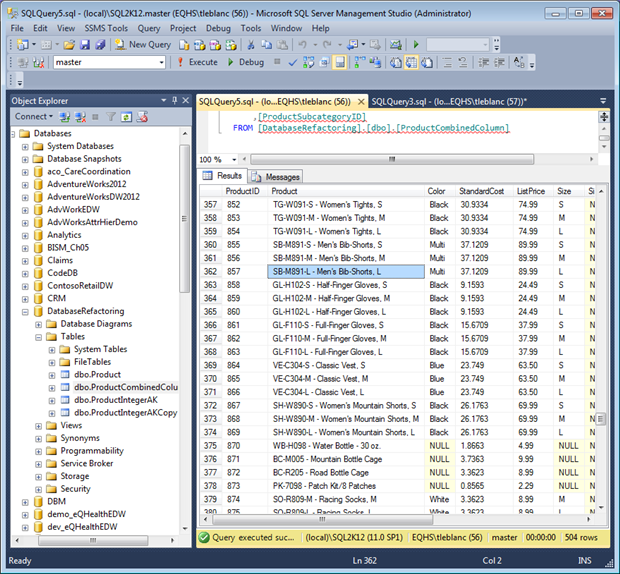
FOREIGN KEY(SubcategoryID)

REFERENCES dbo.ProductSubcategory (SubcategoryID)

GO

**Script 8**

The data looks as shown in Figure 1.



**Figure 1**

Querying a composite column such as Product can be quite a challenge. The only way to return information about all types of gloves, for example, is to use an inelegant LIKE% search. The less stringent the checks to ensure that data in the Product column *always* follows exactly the "**AA-ANNN[-A] - Description of n characters**" format, the more elaborate the search algorithm will need to be in order to guarantee it will always return the expected rows.

The fix is to refactor the table to separate the values into distinct columns, such as ProductNumber (Business Key) and ProductName. Having done this, a view or T-SQL query can combine the two columns for drop downs or reporting, if needed.

USE DBRefactor;

GO

ALTER TABLE dbo.ProductCombinedColumn

ADD ProductNumber VARCHAR(25) NOT NULL DEFAULT 'NA';

ALTER TABLE dbo.ProductCombinedColumn

ADD ProductName VARCHAR(100) NOT NULL DEFAULT 'NA';

GO

-- transpose data

UPDATE dbo.ProductCombinedColumn

SET ProductNumber = RTRIM(SUBSTRING(Product, 1,

CHARINDEX(' - ', Product) - 1)) ,

ProductName = RTRIM(SUBSTRING(Product, CHARINDEX(' - ', Product) + 3,

LEN(product)));

GO

-- Drop Index on Product Column

DROP INDEX AK\_Product ON dbo.ProductCombinedColumn;

GO

-- Drop Product Column

ALTER TABLE dbo.ProductCombinedColumn

DROP COLUMN Product;

GO

-- Create new Alternate Key

CREATE UNIQUE NONCLUSTERED INDEX AK\_ProductNumber

ON dbo.ProductCombinedColumn

( ProductNumber ASC );

GO

**Script 11**

If the user still needed the Product column, either temporarily or permanently, we could add a calculated column to give the presentation a standard feel, while still keeping editable data more normalized.

Of course Script 11 corrects only the composite column problem. Firstly, there is other possible refactoring work required on this table, depending on the exact business requirements. I am using AdventureWorks data for this example, for simplicity, but in reality the ProductNumber column might require further refactoring to achieve proper normalization, for example storing the product type code (TG-, SB-, and so on) in a separate column called ProductType.

Secondly, these is a lot of work still needed to find and fix all references to the old column in stored procedures, user-defined functions, and so on, along with developer code and reports. Use of a calculated column, as suggested above, we mean that we'd only have to refactor writers, and not readers.

Thirdly, we need to consider carefully how to implement this refactoring with minimum risk and minimum disruption of users. Generally such scripts would contain IF EXISTS… guard clauses, for example to prevent the script from running on a database that already has the ProductNumber column.

IF EXISTS ( SELECT 1

FROM information\_schema.COLUMNS

WHERE table\_schema = 'dbo'

AND TABLE\_NAME = 'ProductCombinedColumn'

AND column\_Name = 'ProductNumber' )

PRINT 'the column exists'

ELSE

ALTER TABLE dbo.ProductCombinedColumn

ADD ProductName VARCHAR(100) NULL;

Also, with a change like this, or an even more complicated refactoring, such as a table split, we need to consider whether we will need to take any dependent applications offline for a period, to make the changes to the database and application code simultaneously, or whether there is a way to make the database change without breaking any dependent applications.

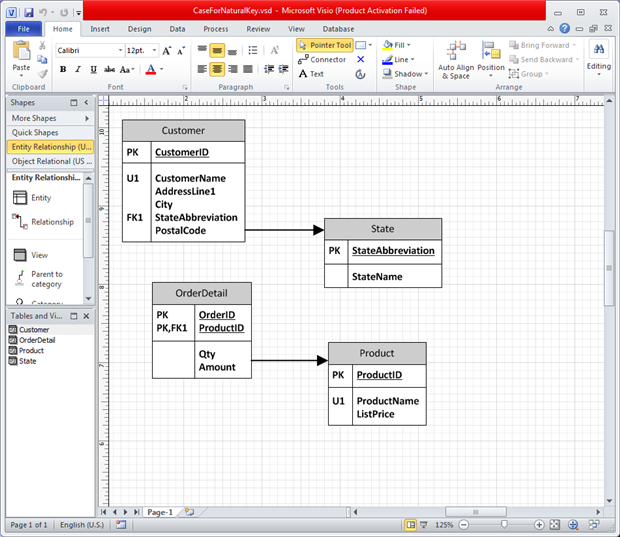
Making an "online" change is sometimes possible, though is much easier and more secure if the design allows the application to access the database through an interface of stored procedures, functions and views, rather than having direct table access. In the former, case we can attempt a strategy referred to as "[feature toggling](https://www.simple-talk.com/sql/database-administration/database-branching-and-merging-strategies/?utm_source=simpletalk&utm_medium=weblink&utm_content=defusingdbs)", whereby the existing stored procedure becomes simply a proxy object. The application continues calling the proxy, which in turn calls a new stored procedure (either renamed or in a different schema), which actually accesses the tables and does the work of providing any necessary defaults, and filtering the returned data into a form the application can accept, until such point as we can update the application to use the new columns.

**No Foreign Keys to Enforce Referential Integrity**

The usual response I get from a developer when I suggest the need to enforce table relationships through Foreign Key constraints is "*we will enforce this in the application*".

However, the application is usually not the only source of data that is inserted into these tables. Without Foreign Key constraints, data will get into a table with a key reference that does not have a key value in the referenced table. Also, data rows can be deleted from the referenced table without checking for existing references to the row in the referring table.

For example, Figure 2 shows a relationship between Customer and State through the StateAbbreviation column. Also, the transaction table OrderDetail has a relationship with Product through the ProductID. By enforcing the relationship through a Foreign Key, the OrderDetail table cannot have ProductID values without a matching ProductID in the Product table. Also, a row in the Product table cannot be deleted if a related row(s) is in the OrderDetail table with that ProductID (assuming NO\_ACTION is used in Foreign Key definition). The Foreign Key relationship can help by making sure the columns in the related tables have the same data type. If not, you get a warning during the creation of the Foreign Key.



**Figure 2**

Of course, Foreign Key constraints offer other advantages as well, such as help the Query Optimizer to come up with a better query execution strategy.

**Poor column sizing or naming**

These problems usually occur due to inadequate requirements gathering initially, or a failure to follow a standard for database naming, or due to implementing a new standard because one did not exist before this point in time.

**Changing the Size of a Column**

When a column size is no longer large enough for new or updated data, we have to increase the size or add another column for spill over. This requires that the several columns are then combined to get the correct value, thereby presenting a maintenance nightmare.

A common cause of this refactoring are string columns that initially stored values no longer than 10 characters, say. If the column originally used a CHAR data type, the developer probably defined it as small as possible, *i.e.*CHAR(10), since any unused space in this type is padded with trailing spaces, which they wanted to avoid. Sometime later the data in this column needs to be more descriptive, and you need to expand the size to accommodate 20-30 characters. At this point, you may choose to switch to a VARCHAR data type, and initially sizing the column to 30 or 40 characters, to accommodate growth. This does not have much impact on the size of the table since the size depends on the inserted value, plus an additional couple of bytes for overhead.

Script 12 increases the size of our ProductNumber column to VARCHAR(50).

ALTER TABLE dbo.ProductCombinedColumn

ALTER COLUMN ProductNumber VARCHAR(50) NOT NULL;

GO

**Script 12**

Again, the change is easy but the real work comes in tracking down and refactoring all stored procedures, user-defined functions, triggers and application code that reference the column to accommodate the length change, as well as variable or temporary tables. For example, if we forget to modify our current stored procedure definition (Script 7) and it receives a product number with 30 characters, the column value will be truncated and no product returned or possibly even the wrong product.

ALTER PROCEDURE dbo.ProductIntegerAK\_Get\_ProductID

@txtProductNumber VARCHAR(50)

AS

SELECT ProductID

FROM dbo.ProductIntegerAK

WHERE ProductNumber = @txtProductNumber

GO

**Script 13**

**Changing a Column Name**

Renaming a column does not cause as much problems with the database structure as it does the code. Script 14 shows how to rename a column.

EXEC sp\_RENAME 'dbo.Product.Name', 'ProductName', 'COLUMN'

**Script 14**

You actually get a nice warning:

Caution: Changing any part of an object name could break scripts and stored procedures.

Now, like the examples above, the DBA and developer have to go through their code and find all dependences and change the references to the column name.

**Summary**

Databases have to be designed correctly from the start because they are, in effect, a formal description of the business data. Paradoxically, a well-designed database is then far better suited to the inevitable subsequent refactoring. The design will be strict enough to enforce current business requirements and ensure data integrity, but flexible enough to change gradually, as business requirements evolve.

The challenge for any database developer is to be able to make changes to the database in the light of evolving understanding of the business. If your database can't be changed in the light of changing business requirements, or changes to the legislative framework that governs your business, then there's surely something wrong with its design.