SQL For .NET Programmers

Moving beyond SELECT, INSERT, UPDATE, DELETE without becoming a DBA

D. M. Bush

Second Edition

SQL FOR .NET PROGRAMMERS

By D. M. Bush

Version 2.0

Table of Contents

Who is This Book for?

This book is for programmers. There are plenty of really thick books that go into all the details about each of the subjects I am going to cover in this book. But if you spend most of your day in languages like C#, those books have way more information than you need. The result is, you'll probably never find the information that would really help you in your day to day programming life.

Let's face it, you know you should know more about SQL than you do, but right now you don't even know what you don't know.

And so this book is intended to be a survey of SQL. Lots of short little chapters that tell you just what you need to know to be productive and doesn't wade you down in every detail of every syntax quirk.

This isn't to say that there aren't tips along the way.

I also want to mention here that while this book isn't big, it is dense. I believe in concise writing. I write just like I code. In as few lines as possible.

I used to work for a training company as a "Programmer who knows how to teach." In one of my classes, one of my students walked out for a few minutes to take a phone call. When he came back, we had moved on to another topic. As he realized he had just missed a huge hunk of important information he said, “You don’t want to walk out of your class, or you’ll miss something important.”

My intention is to provide a lot of value in as little space as possible. You don’t have time to read stuff you’ll never use and providing a book with fluff in it just to get the page count up would only serve as a marketing gimmick.

So, if you are looking for the fastest way to learn the essence of SQL, read on. If you were looking for something with more detail, or more pages, please return this book.

How to Use This Book

This book was written the same way I would teach this material, so the best way to use this book is to read it in order. At least the first time.

Beyond that, I would hope that you would also try the commands along the way. I could have written exercises, but it turns out that the exercises would have looked a lot like the chapters. So, pretend the chapters are the exercises and try stuff out along the way. The only way you'll really learn this is if you use it.

The code is presented using a mono spaced font. Anywhere I use a keyword, I CAPITALIZE it.

Once you've been through this book, you'll probably want to hang onto it as a quick reference to lookup some of the more complex topics until they stick.

Why Bother?

If you are a developer like I am, you've probably gotten by with pretty simple SQL for quite a while. In fact, my experience has been that developers don't get much past SELECT, INSERT, UPDATE, and DELETE with the associated WHERE and ORDER BY clauses that are necessary to get anything done as a developer.

But there are times when knowing more would be helpful. So I thought I'd write a short series on helpful SQL for developers. Keep in mind, I'm a developer, not a DBA. I'm sure there may be better ways of doing some of the things I suggest. However, everything I suggest will be better than trying to do it all from your C# or VB.NET code.

Let's look at some of the reasons a developer should learn SQL:

- If you only use the statements above, your code is almost certainly making more trips to the database than it should, or needs to. Simply put, the performance of your applications is suffering.
- Prior to ASP.NET, and, to a lesser extent, with it, you open yourself up to the possibility of SQL injection attacks because you are only using the statements above.

- You are probably avoiding a good three-tiered architecture because it practically requires you to write stored procedures. This is making your code harder to maintain.
- You are probably using the SqlDataObject for your data binding. As a result, you are frustrated with the errors this is causing you in your updates that you have very limited ways of tracking down.
- By refusing to learn the DDL portion of SQL, you are limited in how you can version your database. If you learn DDL, you'll be able to version your database schemas in a similar manner to how you version your source code.
- Some of the frameworks we use as programmers require we know this information.
- By learning SQL, your SQL will become less of a hack, making this part of your development effort more efficient.
- Even if you have a DBA who is responsible for writing this code for you, wouldn't it be helpful to be able to speak his language? I've been programming for 28 years now and I've only met one DBA who knows more than I do about SQL. Considering that I would not consider myself an expert in SQL this reflects poorly on the DBAs that I've had to work with. What this means in practical terms is, you are a lot more likely to get the SQL code you want into your database if you can just hand it to your DBA and say, "Do this," rather than asking for a stored procedure that does something. Code is the best specification.
- Even if you are using Entity Framework, or some other ORM system, knowing how SQL works under the hood

will allow you to better take advantage of these tools. Most tools in the programming world are like this. They are most effective when you understand what they are doing for you rather than blindly letting them do whatever it is they do.

- Having a firm grasp of SQL will make you a more valuable programmer. This is the best reason of all.

Keep reading as I introduce some SQL that you can use in your daily programming life.

Running SQL from .NET

For the remainder of this book, I'll just be giving you the straight SQL you need to run to make things happen. It will be best to just run the SQL inside SQL Management Studio. But there are times when what we really want to do is to run the SQL code from our .NET code.

The boiler plate code that you can use for this is:

```
var connection =
    new SqlConnection(connectionString);
var command =
  new SqlCommand(sqlToExecute, connection);
connection.Open();
command.ExecuteNonQuery();
connection.Close();
```

Where `connectionString` is the standard connection string you would always give SqlConnection and `sqlToExecute` is the SQL code you want to execute.

CREATE/DROP Database

I thought it would be appropriate, as we embark on SQL for Programmers, to start at the beginning. And while most of our production databases are already created for us, the programmer is the one who is responsible (or should be responsible) for creating databases for testing purposes. Anyhow, at some point you will need to know how to programmatically create and delete a database and you'll be glad you saw this information.

Several years ago, this is exactly where I was. I wanted to create a database programmatically so that I could test a set of modules I had written. I wanted to test the installation process as well as the functionality once the modules were installed into the system.

To do that, I needed to be able to drop a database if it existed and then recreate it so that I could re-install the application and the functionality I was testing.

To create a database, the basic syntax is:

```
CREATE DATABASE dbName
```

If you need a more complex CREATE Database statement, it's just a matter of using SQL which I will leave for you to use your favorite search engine for since this is a book on the basics and is not meant to cover the intricate details of every possibility of each command.

To drop a database, you can't just issue a DROP Database command. You'll need to make sure no one is connected to the database and disconnect all of the connections. Here is the string you will need to do that:

```
ALTER DATABASE dbName
 SET SINGLE_USER
 WITH ROLLBACK IMMEDIATE;
DROP DATABASE dbName;
```

The first statement puts us in single user mode so that we can be sure the command that is executing the SQL is the only that has access to the database. If you don’t do this, your DROP statement will fail.

CREATE TABLE

Once you have your database created, you'll want to get some tables into it. To do this, you'll need the CREATE TABLE statement.

```
CREATE TABLE tableName
  (ColumnName ColumnType(size)
   [IDENTITY(seed,increment)]
   [NULL|NOT NULL
     [DEFAULT defaultData]]
   [PRIMARY KEY]
   [comma delimited column statements here]
  )
```

A column statement is in the form of:

```
ColumnName ColumnType(size)
      [NULL|NOT NULL [DEFAULT defaultData]]
      [Primary Key]
```

Where ColumnType is one of the various data types available in T-SQL

The datatypes you are most likely to use are:

- Int

- varchar(size) or nvarchar(size)
- datetime
- decimal(precision,scale)
- money

If you want the field to hold NULL values, you can leave the NULL or NOT NULL qualifier out, but if you want to make sure the field always holds some sort of valid data, you'll want to include the NOT NULL statement after your type and size information.

And if the field is the primary key for the table, include the Primary Key statement at the end of the column description.

If you want the field to auto-increment, you'll want to include the identity information. Assuming you want to start at 1 and increment by 1, your SQL will look like:

```
IDENTITY(1,1)
```

To put this all together in a typical CREATE TABLE string, here is what a typical CREATE TABLE string would look like for a Users table:

```
CREATE TABLE [dbo].[Users]
  ([UserId] [int] IDENTITY(1,1)
      NOT NULL PRIMARY KEY,
   [UserName] [nvarchar](100) NOT NULL,
   [FirstName] [nvarchar](50) NOT NULL,
   [LastName] [nvarchar](50) NOT NULL,
   [Email] [nvarchar](256) NULL,
   [DisplayName] [nvarchar](128)
      NOT NULL DEFAULT('')
 )
```

There is more to the CREATE TABLE syntax, but this should get most programmers as far as they need to go. Once again, if you need more information there is always the Internet.

ALTERing the TABLE

One of the main problems we seem to be stuck with at this point in programming history is, how do we version our databases?

You could put the whole database into version control, I guess. But that could end up being a lot of data to create a delta for each time.

You could just version the structure. But then you are left with importing data from your current system, which may or may not work.

So most shops either ignore the problem entirely (not a very good idea, if you ask me), or they require every structural change to be recorded in a SQL script so that by running a series of scripts, you can recreate the database to any point in time.

If you are working on a commercial project, or any other project that has to be deployed to multiple locations, you need to at least create delta SQL scripts so that the new location can get the new data.

And creating these delta scripts isn't as hard as it seems, once you learn the basics. But there are now several tools available that do the bulk of the work for you.

There are several things you might need to change in your database as time passes, but the one you will probably need to do the most is add, remove and modify fields/columns. You do this with the ALTER TABLE command.

If all you need to do is to add a column to your table, the command is quite simple and looks a lot like the CREATE TABLE statement we looked at in the previous chapter.

```
ALTER TABLE tableName
    ADD newField datatype(size,precision)
      etc...
```

To remove a field, you use the DROP statement

```
ALTER TABLE tableName
    DROP newField
```

To change a field definition

```
ALTER TABLE tableName
    ALTER COLUMN newField
      dataType(size,precision) etc...
```

As far as I know, there is no clean SQL way to change a column name. You'd have to

1. ADD a new field with the new name and type information
2. Issue an UPDATE command to move the data from one field to the other
3. DROP the old field.

Many times we'd like to ADD several fields, drop several fields and change the data type information all at one time. This is where most of the documentation you can find online for the ALTER TABLE command falls short.

To add multiple columns:

```
ALTER TABLE tableName
    ADD newField datatype(size,precision)
      etc... ,
    newField2 datatype(size,precision)
      etc...,  etc...
```

To drop multiple columns:

```
ALTER TABLE tableName
     DROP newField,
     newField2
```

So you’d think you could change multiple columns the same way, right?

WRONG!

To alter multiple columns, you will need to issue multiple ALTER TABLE statements.

You also cannot ADD and DROP in the same ALTER statement. You will need one ALTER TABLE statement to ADD new columns and another ALTER TABLE statement to DROP all of your columns and yet another ALTER TABLE statement for each column you want to modify.

Basic SQL Commands

Before we get into the specifics of the commands that can be used within a stored procedure, I think it would be helpful to review some of the more basic commands that we can use that don't really need a whole lot of discussion.

I know I said I wasn't going to spend a lot of time on the basics, but my research indicates that a lot of people are searching for basic SQL commands and very few sites are addressing that issue specifically.

So, if this is unhelpful to you, just move along to the next chapter.

Basic SQL Commands

- SELECT
    - The SELECT statement returns a set of records from a table or view. It can be used in combination with JOIN to return a set of records that are a mashup of records from multiple tables but the end result from a programming perspective is that it looks like it came from one table. The set of records returned can be narrowed by using the WHERE clause and the

order that they are returned can be set by using the ORDER BY clause.

- Most common form:

```
SELECT field1, field2, etc
  FROM tableName
  WHERE conditions
  ORDER BY listOfFields
```

- INSERT
    - INSERT adds new records to the table. This can be used in combination with a sub-SELECT to insert records from another table into the current table.
    - Most common form:

```
INSERT INTO
  tableName(field1,field2)
  VALUES(value1,value2)
```

- UPDATE
    - This is most often used in combination with the WHERE clause to update a record in a table.
    - Most common form:

```
UPDATE tableName
  SET field1=value1,
      field2=value2
  WHERE condition
```

- DELETE
    - When you need to delete a record in your table, or a set of records, you will use the DELETE statement.
    - Most common form:

```
DELETE FROM tableName
  WHERE condition
```

- WHERE

    - The WHERE clause specifies what set of records the operation should be performed on. The most common operators are: =, <>, <, >, IN, and LIKE
- IN
    - IN is used with the WHERE clause to determine if the field is equal to any of the items in a list. We cover this in more depth in a couple of chapters.
- LIKE
    - Most are familiar with the common form, LIKE '%content%' to determine if 'content' is in the field. But there are other useful forms of this operator as well. We cover this operator in more detail in the next chapter.
- ORDER BY
    - Most often used with the SELECT statement (and I really can't think of any other place it is used), this controls the order the records are returned. Most often this is a list of fields. By default, it lists everything in ascending order. If you want the items in descending order, use the DESC modifier.

Finding a String

Many times in our queries, we aren't looking for an exact match. We are looking for one string that exists in another. There are a couple statements available to us that will allow us to do this.

The first of these is the LIKE statement, and if you are familiar with DOS or the Linux equivalent, this should look familiar to you.

The most common usage of LIKE looks like this:

```
SELECT * FROM someTable
     WHERE fieldName LIKE '%stringHere%'
```

Note the % inside the single quote marks.

This statement is telling SQL to find all of the rows from someTable where the contents of fieldName has the string 'stringHere' in it.

You could also do something like:

```
SELECT * FROM someTable
     WHERE fieldName LIKE 'stringHere%'
```

which would find all the rows where fieldName started with 'stringHere'

You can also use the following pattern matching characters:

Symbol	- ***Description***
_	**Look for any single character**
[]	**Look for any single character listed in the set**
[^]	**Look for any single character not listed in the set**

`fieldName LIKE '[abc]%'` - Looks for anything that starts with a, b, or c.

`fieldName LIKE '[^abc]%'` - Looks for anything that doesn't start with a, b or c.

If you need to find a string that includes one of these symbols, you'll need to ESCAPE it and you'll need to define the escape sequence.

So to look for a % in the middle of the string you would use:

```
SELECT * FROM someTable
     WHERE fieldName LIKE '%%stringHere%'
     ESCAPE '%'
```

This will look for all rows where the fieldName starts with the string '%stringHere' because we've told it that % is the escape character and we've used %% twice in our search string.

Finding IN a List

In the last chapter, we looked at looking for content that was LIKE other content. While this has its uses, it is limited in its ability to find more than one pattern. So what if we want to find a record that matches more than one set of data?

That is what the IN clause is for.

The most common use of the IN clause is:

```
SELECT * FROM myTable
     WHERE fildName IN
      (comma,Separated,List)
```

So if you were looking for the records whose ID were equal to 1, 3, or 5, your query would look like:

```
SELECT * FROM myTable
     WHERE ID IN (1,1,5)
```

You could also use this with strings:

```
SELECT * FROM myTable
     WHERE firstName IN
       ('Dave','George','Frank')
```

The final form is using a sub-select. I should warn you that most DBAs frown on this form since it does have performance issues. Later on we will look at the alternative, JOINs. JOINs perform MUCH faster.

One of the main problems with a sub select is that the sub-select runs for each row in the main select. If you only have a few rows, maybe even one row, then it won't make much difference which you use. But when you are trying to retrieve a lot of data, JOIN is going to be your friend.

Anyhow, here is the syntax for sub-selects just to be complete:

```
SELECT * FROM myTable
     WHERE firstName IN
      (SELECT firstName FROM myOtherTable
         WHERE someOtherCondition)
```

I recently used this in combination with NOT to find fields in the main table that didn't have corresponding records in the sub-select table.

```
SELECT * FROM myTable
     WHERE firstName NOT IN
      (SELECT firstName FROM myOtherTable
         WHERE someOtherCondition
```

It would have been better if I had done that with a JOIN. But for a one off, it got the job done.

JOINS

I hate JOINS. As a programmer, I find them to be one of the most confusing concepts to get my head around. Or at least I did. Most of the time, I would much rather have pull out a trusty sub-select and just get on with my day. If you are anything like me, you probably find them confusing as well. But they don't have to be.

JOINS are just a lot more efficient than sub-selects at working with data from multiple tables. So if you are going to be working with databases, you need to make an effort to learn how JOINs work.

The good news is, if you work for a really large company where the performance is going to matter, you can probably just hand your DBA your crappy sub-select code and they can optimize it for you.

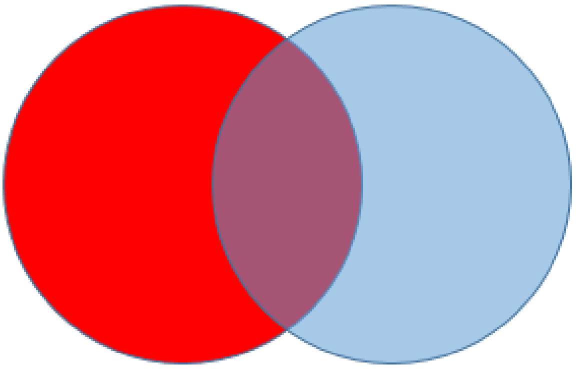
But we are programmers. We may not be able to write the absolutely most efficient SQL in the world, but why would we use syntax that we know is going to need to be optimized if we can write it at least half way performant in the first place?

And so, we need to learn how to use JOIN.

But, before we get into syntax, I think it would be a good idea if we started with asking, what problem does JOIN solve?

One of the best explanations I've heard of how JOINs work and what problem the solve comes from Set Math. You know. All that Union and Intersection stuff we learned long ago in Algebra. Only with JOIN, we are applying all of that same set theory to tables.

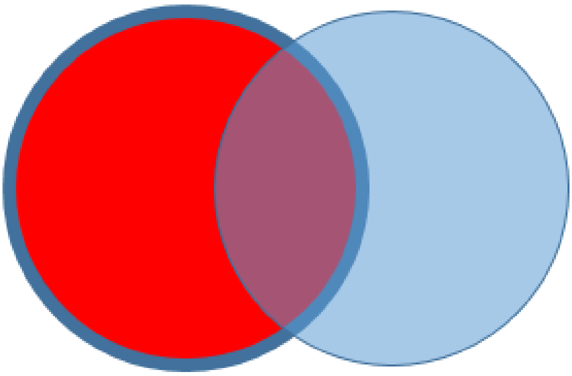
Typically, what we want to know is this. Given two tables, give me all of the records that are common between the two tables. This is what is referred to as an INNER JOIN. In common set theory, this is typically illustrated like this:



Where the middle section is the results of the INNER JOIN.

The next most popular type of JOIN is when we want all of the records from the left, regardless of if they match the records on the right or not. But any records on the right that do match should be matched up with the records on the left.

This is often illustrated by a diagram that looks like this:



And the name of this kind of JOIN is a LEFT OUTER JOIN

Frequently, when we want all of the records from one table and the matches from the other, we place the table we want all of the records from on the left. But it is possible to put them on the right instead. This is called a RIGHT OUTER JOIN. Just imagine the diagram flipped around for what that would look like visually.

But, what if you want all of the records from both tables? There are two things you might mean by that.

If you want all of the records from the left regardless of match and all of the records from the right regardless of match, but where there is a match you want the records matched up. What you are looking for is a FULL OUTER JOIN.

Or, maybe what you want is all of the possible combinations. This is hard to illustrate from pictures. But what you'll end up with is a result that has the number of rows on the left multiplied by the number of rows on the right. While this is not used very often, it is useful in a limited number of circumstances. One that I find particularly useful is when I need to generate test data. This is called a CROSS JOIN.

All other types of JOINs are derived from these five basic JOINs.

Of course, it all sounds so very simple. But what does it look like with data?

LEFT OUTER JOIN

It would be tempting to start the coding section with either an INNER JOIN or a CROSS JOIN but I think JOINS are best understood from a coding perspective if we start with a LEFT OUTER JOIN because this is the JOIN that most closely resembles something we already know. A sub-select.

Give that Table_1 looks like this:

ID	Name	Color
1	Dave	Green
2	Bob	Blue
3	Julie	Red

And Table_2 looks like this:

ID	Address	ParentId
3	21 Street	1
4	43 Street	1
5	X Street	3
6	Z Street	4

If we were to join these two tables using a sub select,

```
SELECT [ID], [Name], [Color],
  (SELECT Address FROM Table_2
  where ParentID = Table_1.Id) AS Address
  FROM [Table_1]
```

You will quickly find out just how limiting Sub Selects can be. Because sub selects can only return one row. Also, note it would take multiple sub-selects to return multiple columns. So, even though this is the closest thing we have to a LEFT OUTER JOIN, even the sub-select is useless in this case.

We could, of course, flip the tables, but if we want all of the rows from table_1, that won't work either.

What we want to do instead is a LEFT OUTER JOIN

```
SELECT Table_1.ID, Table_1.Name,
 Table_1.Color, Table_2.ParentId,
 Table_2.Address, Table_2.ID AS Table2ID
  FROM Table_1 LEFT OUTER JOIN
    Table_2 ON
      Table_1.ID = Table_2.ParentId
```

As we described above, this will return all of the rows from table_1 and all of the rows that match in table_2

ID	Name	Color	ParentId	Address	Table2Id
1	Dave	Green	1	21 Street	3
2	Dave	Green	1	43 Street	4
3	Bob	Blue	NULL	NULL	NULL
3	Julie	Red	3	X Street	5

Note two rows for Dave (ID = 1)

All you need to remember is that a LEFT OUTER JOIN returns all of the rows on the LEFT and matches everything up.

RIGHT OUTER JOIN

The RIGHT OUTER JOIN does the exact opposite of the LEFT OUTER JOIN. It takes everything from the right table and matches it up to the table on the left.

Using the example from the previous chapter

```
SELECT Table_1.ID, Table_1.Name,
 Table_1.Color, Table_2.ParentId,
 Table_2.Address, Table_2.ID AS Table2ID
  FROM Table_1 RIGHT OUTER JOIN
    Table_2 ON
      Table_1.ID = Table_2.ParentId
```

Produces this result.

ID	Name	Color	ParentId	Address	Table2ID
1	Dave	Green	1	21 Street	3
1	Dave	Green	1	43 Street	4
3	Julie	Red	3	X Street	5
NULL	NULL	NULL	4	Z Street	6

Similar, but not quite the same.

Notice that this time we attempted to match to ID 4 but it doesn’t exist in Table_1

INNER JOIN

But what if you don't want those NULL fields showing up because there wasn't a match? Both the LEFT and RIGHT OUTER JOINS give the NULL fields. What we need for this case is the INNER JOIN.

```
SELECT Table_1.ID, Table_1.Name,
 Table_1.Color, Table_2.ParentId,
 Table_2.Address, Table_2.ID AS Table2ID
  FROM Table_1 INNER JOIN
    Table_2 ON
      Table_1.ID = Table_2.ParentId
```

And the result it gives us ends up looking like this:

ID	Name	Color	ParentID	Address	Table2ID
1	Dave	Green	1	21 Street	3
1	Dave	Green	1	43 Street	4
3	Julie	Red	3	X Street	5

Notice how only the records that match from both tables show up.

FULL OUTER JOIN

Maybe you want the records from both sides but you want all of the records from both sides. This, of course, will cause the NULL fields to show up again, but they will show up on both sides because we want all of the records from both tables.

```
SELECT Table_1.ID, Table_1.Name,
 Table_1.Color, Table_2.ParentId,
 Table_2.Address, Table_2.ID AS Table2ID
  FROM Table_1 FULL OUTER JOIN
    Table_2 ON
      Table_1.ID = Table_2.ParentId
```

Produces this result:

ID	Name	Color	ParentID	Address	Table2ID
1	Dave	Green	1	21 Street	3
1	Dave	Green	1	43 Street	4
2	Bob	Blue	NULL	NULL	NULL
3	Julie	Red	3	X Street	5
NULL	NULL	NULL	4	Z Street	6

CROSS JOIN

As mentioned before, the CROSS JOIN gives us all combinations.

```
SELECT Table_1.ID, Table_1.Name,
 Table_1.Color, Table_2.ParentId,
 Table_2.Address, Table_2.ID AS Table2ID
  FROM Table_1 CROSS JOIN
    Table_2 ON
      Table_1.ID = Table_2.ParentId
```

And the result looks like this:

ID	Name	Color	ParentId	Address	Table2Id
1	Dave	Green	1	21 Street	3
1	Dave	Green	1	43 Street	4
1	Dave	Green	3	X Street	5
1	Dave	Green	4	Z Street	6
2	Bob	Blue	1	21 Street	3
2	Bob	Blue	1	43 Street	4
2	Bob	Blue	3	X Street	5
2	Bob	Blue	4	Z Street	6
3	Julie	Red	1	21 Street	3
3	Julie	Red	1	43 Street	4
3	Julie	Red	3	X Street	5
3	Julie	Red	4	Z Street	6

GROUP BY

If GROUP BY and aggregate functions have been as confusing for you as they were for me, you are going to love this chapter. And here is the first quick hit tip about GROUP BY: there is practically no reason to use this on a single table. But its usefulness really shows when you have a one to many OUTER JOIN.

Let’s start off with a classic problem I think most of us can relate to if we haven’t actually coded for it in the past. Two tables. Orders and OrderItems. The structure of our Orders table will contain:

Field	Type
Id	Auto incrementing Int
CustomerId	Int
ShippingCost	money

And the structure of our OrderItems table will look like this:

Field	Type
Id	Auto incrementing Int
OrderId	int
ItemTotal	money

And so we are all on the same page, here is the data we will use for this example:

Orders		
Id	CustomerId	ShippingCost
1	1	20
2	1	30
3	1	40
4	2	15
5	2	15

OrderItems		
Id	OrderId	ItemTotal
1	1	100
2	1	150
3	2	125
4	2	50
5	2	75
6	3	100
7	4	33
8	4	17
9	5	30

Now we can JOIN these using the syntax from the previous chapters on JOIN.

```
SELECT Orders.ID, Orders.CustomerId,
 Orders.ShippingCost, OrderItems.Id
 AS ItemId,
 OrderItems.ItemTotal
  FROM Orders LEFT OUTER JOIN
    OrderItems ON
      Orders.ID = OrderItems.OrderId
```

Which will result in

Id	CustomerId	ShippingCost	ItemId	ItemTotal
1	1	20	1	100
1	1	20	2	150
2	1	30	3	125
2	1	30	4	50
2	1	30	5	75
3	1	40	6	100
4	2	15	7	33
4	2	15	8	17
5	2	15	9	30

And now, we can start to do something meaningful with GROUP BY.

The first thing we are going to want to do is that we are going to want to find out what the total of the ItemTotal is for each order. This is pretty easy. We want to SUM(ItemTotal). Since we want to know the total of each order, we will GROUP BY Orders.Id

Since we aren't aggregating any of the other fields, we will need to drop those from our SQL statement. This leaves us with the following SQL.

```
SELECT Orders.ID,
 SUM(OrderItems.ItemTotal) AS OrderTotal
  FROM Orders LEFT OUTER JOIN
    OrderItems ON
```

```
      Orders.ID = OrderItems.OrderId
  GROUP BY Orders.Id
```

Which leaves us with the following summary of data:

Id	OrderTotal
1	250
2	250
3	100
4	50
5	30

There are a couple of ways that we could pull in the shipping cost for each order. We just need an aggregate function that will show us the one number for the order. The easiest way would be to add the ShippingCost as a field and add it to the GROUP BY. And just to keep things neat, we will add an ORDER BY clause to keep things in ID order.

```
SELECT Orders.ID, Orders.ShippingCost,
 SUM(OrderItems.ItemTotal) AS OrderTotal
  FROM Orders LEFT OUTER JOIN
    OrderItems ON
      Orders.ID = OrderItems.OrderId
  GROUP BY Orders.Id, Orders.ShippingCost
  ORDER BY Orders.Id
```

This results in the following data:

Id	ShippingCost	OrderTotal
1	20	250
2	30	250
3	40	100
4	15	50
5	15	30

We can get the same result using:

```
SELECT Orders.ID,
 SUM(Distinct Orders.ShippingCost),
 SUM(OrderItems.ItemTotal) AS OrderTotal
  FROM Orders LEFT OUTER JOIN
    OrderItems ON
      Orders.ID = OrderItems.OrderId
  GROUP BY Orders.Id
  ORDER BY Orders.Id
```

Note the SUM(Distinct …) and that we removed the ShippingCost from the GROUP BY.

SUM(Distinct) tells SQL to only add the unique values. And since all the values are the same for the group, we get the one and only value.

If we had left out the Distinct part, it would have given us an entirely different report with the shipping cost multiplied by the number of items in the order.

There is a problem with SUM(Distinct) that you need to be aware of. It is SUMming distinct values, not distinct rows, so if you are grouping by a field that will result in two rows that should be summed but they have the same value in them, your SUM is not going to give you the value you would expect. If you run into this case, the solution is to break down the problem into multiple views that you can JOIN together once you have the correct answers for each part.

HAVING

One of the biggest confusions people tend to have with SQL is the difference between WHERE and HAVING.

Here's what you need to keep in mind. WHERE clauses are processed during the selection of rows prior to any grouping you might do. HAVING selects the resulting rows after the GROUP BY. If you aren't using grouping in your SQL, you don't need HAVING at all.

So a HAVING clause is going to look a lot like a WHERE clause but it will probably be based on some aggregate function.

Adding HAVING to the code we wrote in the previous chapter:

```
SELECT Orders.ID,
 SUM(Distinct Orders.ShippingCost),
 SUM(OrderItems.ItemTotal) AS OrderTotal
  FROM Orders LEFT OUTER JOIN
    OrderItems ON
      Orders.ID = OrderItems.OrderId
  GROUP BY Orders.Id
  HAVING SUM(OrderItems.ItemTotal) > 20
  ORDER BY Orders.Id
```

UNIONs

The UNION statement is interesting because it allows us to create one dataset from multiple tables or views.

Here are some basic rules to keep in mind when using UNION:

- The final result of a UNION will be the distinct records from each SELECT that created it. If you want all of the records from all of the SELECTs that created it, use SELECT ALL.
- The first SELECT will determine the names and types of the columns. Make sure all of the following SELECTs have the same number of columns and that the column types are compatible with the types of the first SELECT.
- ORDER BY and aggregate functions work on the final result.
- GROUP BY and HAVING only work in individual SELECT statements

The basic syntax of UNION is quite simple:

```
SELECT field1, field2, field3, etc...
UNION
SELECT fieldA, fieldB, fieldC, etc
```

Or, for UNION ALL

```
SELECT field1, field2, field3, etc...
UNION ALL
SELECT fieldA, fieldB, fieldC, etc
```

One trick I just recently learned about that uses UNION in a very creative way is for INSERTing records into a table.

```
INSERT INTO someTable (field1, field2,
field3)
SELECT 1, 'a', 'ab'
UNION
SELECT 2, 'd', 'ef'
```

Filtering on two rows

I once received the following question from a programmer:

Given the following table:

Name	**Language**
Nikhil	Hindi
Nikhil	English
Kisu	Hindi
Kisu	English
Rakesh	Hindi
Kousik	Bangali

How do I select names of persons who know both Hindi and English?

In this table, that query should return Nikhil and Kisu.

Normally, if it were me, I'd create a Language table and a Name table and use an intermediate table to join them. But the solution would still be about the same. So we'll work with the table we've been given. There is, obviously, a Bangali table to be had, but we only want those who speak English and Hindi.

What we need to do here is make this look like it is two tables—The Hindi table and the English table—and do a JOIN between the two:

```
SELECT [NameLang].[Name]
 FROM [NameLang]
   INNER JOIN
   [NameLang].[Name]
      = [NameLang_1].[Name] WHERE
    ([NameLang].[Language]
      = 'English')
   AND
    ([NameLang_1].[Language]
      = 'Hindi')
```

This pulls all of the names from the table where the person speaks English and then JOINS them to a selection of all the names of people who speak Hindi. What you get back is a list of names of all the people who speak both.

Stored Procedures

While we could go on with the various syntax elements in a SELECT statement, I doubt that it would be as helpful as the items we've already covered. Most of what I use on a day-to-day basis is some combination of what I've shown already. Actually, there is quite a bit more that I've shown you than I actually have to use every day.

Where SQL really starts getting fun, though, is with stored procedures.

But why use Stored Procedures at all?

The historic answer to this question has always been, "speed." Every time you send a query statement to the SQL server from your application, it has to re-compile it. There are ways to compensate for this, but the code you'd need to write to do this is almost harder than just writing the stored procedure to begin with.

However, now that we have LINQ, it is really hard to make the performance argument any more. In fact, the argument of the day is, ".NET Programmers don't need to learn SQL, they should learn LINQ instead." And, if all you ever write is a basic SELECT, INSERT, UPDATE, or DELETE statement, you'd be right. LINQ has all that optimization code built into it, so you won’t get any significant performance gains by moving to a stored procedure. And you could get a performance gain if you actually write a LINQ statement that does everything you need it to do instead of writing some hybrid of LINQ and good old fashion C# or VB.NET code.

However, there are times when using a stored procedure is going to make more sense. The primary one being that your organization has not yet gotten on the LINQ band wagon. Or you are using a version of SQL that does not yet have a good LINQ2SQL driver.

So in the next chapter we’ll start looking at the syntax for stored procedures and some of the things we can do in a stored procedure other than just wrapping our SELECT, INSERT, UPDATE, and DELETE statements.

Stored Procedure Basics

The first thing that makes any procedure a procedure is the ability to define the procedure name and define the parameters that can be passed. This is also true of stored procedures.

The basic structure of a stored procedure looks like:

```
CREATE PROCEDURE procedureName (
 /* Parameters go here */ ) AS
/* body of the procedure
   goes here */
```

If you later needed to change this procedure, you would use this syntax:

```
ALTER PROCEDURE procedureName (
 /* Parameters go here */ )
 AS
 /* body of the procedure
   goes here */
```

If you don’t have any parameters to pass, you can leave out the parentheses like so:

```
CREATE PROCEDURE procedureName
 AS /* body of the procedure
   goes here */
```

You can define parameters to your procedure using comma-delimited lists of @ prefixed names. You can name them whatever you want. Also keep in mind that SQL is not case sensitive. When you define the variables, you will also need to specify the type and, optionally, the size of the variable.

So to pass in a string and an integer, you would pass in:

```
CREATE PROCEDURE procedureName
   @StringVar as VARCHAR,
   @IntVar as Int
 AS
   /* body of the procedure
      goes here */
```

In the example above, the @StringVar would only see the first character.

If you wanted to make the size of the string 50 characters, you would use:

```
CREATE PROCEDURE procedureName
   @StringVar as VARCHAR(50),
   @IntVar as Int
 AS
 /* body of the procedure
   goes here */
```

Then, if you wanted to use those variables as part of your where clause, you would use:

```
CREATE PROCEDURE procedureName
   @StringVar as VARCHAR(50),
   @IntVar as Int
AS
   SELECT fieldOne, fieldTwo,
          etc
   FROM tableName
   WHERE stringField=@StringVar
     AND intField=@IntVar
```

You can find a full list of datatype definitions at the Microsoft site:

http://msdn.microsoft.com/en-us/library/ms187752.aspx

By default, the parameters are input only. I've found that that is all I need. But you may need to return a value through a parameter for some reason. You can do that by defining the parameter as an OUTPUT parameter:

```
CREATE PROCEDURE procedureName
   @StringVar as VARCHAR(50),
   @IntVar as Int
 AS
   /* code that does
     something meaningful */
   SELECT @IntVar = 20
```

Notice you have to assign the output parameter a value.

All stored procedures either return nothing or they return a set of rows. So a better way of writing the stored procedure so that it returns 20 is to use something like this:

```
CREATE PROCEDURE procedureName
   @StringVar as VARCHAR(50),
   @IntVar as Int
 AS
  /* code that does
     something meaningful */
   SELECT 20 as intField
```

where intField could be any meaningful name that could be used as a column name.

SQL IF Blocks

Here's where SQL starts getting interesting. You have the basic concepts of declaring variables, setting up stored procedures, and doing a basic insert, update or delete statement. And if we stopped here, you'd probably be able to do 80% of the work you need to do.

You'd also start to wonder why you should even bother to learn SQL.

But now we tackle the basics of conditional programming in SQL.

The good news is that the two commands you are going to use for conditional programming are pretty simple. The bad news is, all the features and functionality you are used to having in just about any other programming language are gone.

Let's take a look at your basic IF statement, which is simple in any language:

```
DECLARE @someString as VARCHAR
IF @someString='ABC'
   SELECT * FROM someTable
```

Obviously, I'm skipping some steps. But let's assume that @someString was actually passed in. I just declared it so you could see that it is a VARCHAR.

This is about as simple as it gets. If @someString equals "ABC" then go ahead and get the data.

The trick is, what if you want to run multiple lines as part of your condition?

In that case, you'll need a BEGIN and END statement:

```
DECLARE @someString as VARCHAR
IF @someString='ABC'
 BEGIN
   /* do something
     meaningful here */
   SELECT * FROM someTable
 END
```

BEGIN and END are the opening curly brace and closing curly brace of the SQL programming world. A lot more like Basic than C#.

In the real world, we'd be storing data into variables. But, I think we're all programmers enough to be able to look at the general idea.

Where the syntax starts looking a bit messy is when you use ELSE:

```
DECLARE @someString as VARCHAR
IF @someString='ABC'
```

```
  BEGIN
   /* do something
     meaningful here */
   SELECT * FROM someTable
  END
 ELSE
  BEGIN
   /* do something else */
   SELECT * FROM someTable
  END
```

Even with a bit of indenting, it doesn't clean it up much:

But this is what we have to live with if we want to program in SQL.

Next, we'll take a look at WHILE loops.

SQL WHILE

The IF statement we looked at in the last chapter was pretty tame compared to the WHILE construct.

Actually, the main thing you need to keep in mind is that WHILE is all you have. There is no such thing as a FOR loop or a DO WHILE loop. So you have to force WHILE to do those for you.

The basic syntax of WHILE looks like this:

```
DECLARE @someString as VARCHAR
 WHILE @someString='ABC'
  BEGIN
   /* do something
     meaningful here */
   SELECT * FROM someTable
 END
```

So if you want a FOR/NEXT loop, you'll need to write:

```
DECLARE @someInt as int
 SET @someInt = 0
 WHILE @someInt < 20
 BEGIN
   /* useful code here */
   SET @someInt = @someInt + 1
 END
```

and a DO WHILE loop would be something like:

```
DECLARE @someInt as int
 SET @someInt = 0
 WHILE @someInt = 0
 BEGIN
   /* useful code here */
   IF /*some exit condition*/
      SET @someInt = 1
 END
```

Once you learn to substitute those constructs for your normal FOR/NEXT or DO WHILE code, it becomes rather easy to deal with.

Temporary Tables

You may think that you don’t need a temporary table. But if you’ve ever retrieved data from your database or retrieved data from a table and put it in a list of some sort simply to process it further with your code, you need to learn about temporary tables. Because, that's how you do that kind of stuff in SQL.

I’ve also needed to use Temporary Tables to simulate arrays in my stored procedures. You can pass in a comma-delimited list and then have your stored procedure process the list into a temporary table, making the information easier to evaluate.

In the first case, we reduce the amount of data that needs to pass between our database and our code. In the second case, there aren’t any other options to achieve what it is we want to do.

Creating a temporary table looks very similar to creating a regular table. The only difference is that we put a pound sign (#) in front of the name of the table name.

```
CREATE TABLE #ourTable
 (
   ID int,
   fieldTwo varchar(50),
   fieldThree int
 )
```

Then, later in our code, we can use it like a regular table:

```
INSERT INTO #ourTable
 (
   [ID],
   [fieldTwo],
   [fieldThree]
  )
  VALUES
 (
   @ID,
   @fieldTwo,
   @fieldThree
  )
```

As a dotNet programmer there is a small quirk about temporary tables you need to know about. If you return a temporary table from a stored procedure, the wizards will not be able to see it so that they can create your DataTable in your DataSet correctly. My suggested work-around for this is to create a real table and have your stored procedure use that table while you are running the DataTable wizard. Then, once you have the DataTable, change your stored procedure back to using the temporary table.

Then again, if you are still using DataSets and DataTables you probably need to update more than your SQL skills. Have you tried LINQ or Entity Framework yet?

SQL CURSOR

One of the things it took me a while to figure out was how to get a stored procedure to loop through data in a table and pull out specific data I needed.

There are actually two parts to this question. First, you should always try to find some other way of doing whatever it is you think you need to do by processing one row at a time. SQL is a “Set-Based” language and as such it expects you to work with the whole set. That’s what it was designed to do best and that’s why it has functions such as MAX, MIN, SUM, AVG, etc. Many of these functions in combination with a good WHERE clause will get you the information you need.

However, there are times when you’ll need to process the information one record at a time, and this is what the CURSOR was created for.

Before you use a CURSOR, you’ll need to declare it. The basic form looks like:

```
DECLARE cursorName
    CURSOR for
 SELECT field1, field2
    FROM myTable WHERE whereClause
```

Notice that unlike other DECLARE statements in SQL, we did not prefix the variable with an '@'

Once we have the cursor declared, we need to OPEN it to cause it to execute:

```
OPEN cursorName
```

The rest is going to look very similar to what we did back when we were using ODBC. To move to the first record, we have to FETCH the row:

```
FETCH NEXT FROM
   cursorName INTO @field1,
    @field2
```

Obviously, we would have declared @field1 and @field2 ahead of time. Notice that @field1 and @field2 correspond to the order the fields show up in the SELECT statement above. You can call the variables whatever you want, but it is customary to name them the same as the fields you are getting the data from.

There is a special variable in SQL called @@FETCH_STATUS that will be zero (0) as long as the fetch was able to retrieve a row, so you'll want to set up a while loop after your first fetch that checks the status and then processes the data. Right before the end of the while loop you will issue another FETCH.

```
FETCH NEXT FROM
   cursorName INTO @field1,
    @field2
```

```
WHILE
   @@FETCH_STATUS = 0
BEGIN
  /* do something with
    the data here */
  FETCH NEXT FROM
    cursorName
   INTO @field1,
    @field2
END
```

Once you know how, it isn’t that hard to process records in your stored procedures which will allow you to do a significant amount of your data processing on the SQL server and return to the client only the information that the client needs.

SQL CURSOR Performance

In the last chapter I showed how to use the SQL CURSOR to loop through records in a database. In that chapter I mentioned that you want to avoid using the CURSOR if you can because it has performance problems.

There is an alternate way that typically performs better.

```
SELECT TOP 1 @field1=Field1,
   @field2=Field2,
    @key=keyfield(s)
 FROM someTable ORDER BY keyfields(s)
 WHILE @key
    /* appropriate logic
       to verify that there
       is a record */
BEGIN
   /* processing here */
   SET @lastKey=@key
   SELECT TOP 1
     @field1=Field1,
     @field2=Field2,
     @key=keyfield(s)
   FROM someTable
   WHERE @lastKey < keyfield(s)
 END
```

Note: this is a rough outline of the code. The key point is that you are doing a select for the next record up in order from the one you just retrieved and you are doing it with nothing but select statements.

Also, keep in mind that keyfield(s) could be one, or multiple fields and that the field(s) should be unique. Otherwise this won’t work.

If you need to sort in an order other than the primary key, just make keyfield(s) a composite key with the sort order fields first and the primary key last. Ie assuming all of the following fields are strings, keyfield(s) = field1 + field2 + primaryKeyField.

Instead of doing TOP 1, you could also do a select for the MINimum value of the remaining keys.

And it works for most cases.

However, while "everyone knows" you aren’t supposed to use CURSORS because they are so slow, you need to TEST your solution to make sure that in this particular case it really is faster.

But like I stated when I started, my job here is to get you beyond SELECT, INSERT, UPDATE, and DELETE statements so that you can do more on the server side, not to turn you all into DBAs who know where all of these performance gotchas are.

Return Random Records

Several years ago, I had an interesting assignment: Given a specific record, randomly retrieve three related records from the database.

Naturally, as a programmer, I started looking for the SQL random function. Which I found.

But even as I was searching for how to use that function, I was thinking to myself, "How am I going to structure things in such a way so as to randomly select the records once I have the function? It isn't like I can retrieve the records into an array and then select them out. Am I going to have to retrieve them into a temp table?"

And that's when I stumbled onto a slick little trick. The NewID() function.

The NewID() function returns a value that looks something like a GUID. The beauty of this function is that when you use it as a column in your return set and set the order to that column, you end up with a randomly sorted list of records.

So what I thought was going to be lines and lines of SQL code ended up being a rather simple select statement:

```
SELECT TOP 3
   field1,
   field2,
   NewId as OrderId FROM databaseTable
 WHERE @filter=filter ORDER BY OrderId
```

VIEWS

That's quite a lot of syntax so far. And still there are a few more items we need to cover before we can move on to the more practical questions of testing and version control.

While VIEWS may not be something you see a lot of, they really come in handy when used in combination with JOINS because we can create a JOIN in a VIEW and then use the VIEW as though it were a TABLE.

You can, of course, use the designer in SQL Manager Studio to create a view, but this book is all about syntax. How would you create it using a SQL Script?

```
CREATE VIEW [ViewName]
AS
  View Script Goes Here
```

To make a VIEW out of one of our JOINS, it would look like this:

```
CREATE VIEW [dbo].[T1T2View]
AS
SELECT dbo.Table_1.ID AS T1ID,
dbo.Table_1.Name,
 dbo.Table_1.Color, dbo.Table_2.Address,
 dbo.Table_2.ID AS T2ID
FROM dbo.Table_1 INNER JOIN
 dbo.Table_2 ON dbo.Table_1.ID =
   dbo.Table_2.ParentId
```

Later on we could use a select statement to retrieve the data from that view using:

```
SELECT T1ID, Name, Color, Address, T2ID
FROM T1T2View
```

Modifying the view is similar to modifying a stored procedure.

```
ALTER VIEW [ViewName]
AS
     View Script Goes Here
```

And of course dropping a view

```
DROP VIEW [ViewName]
```

You might be thinking, “I could do something similar with a stored procedure.”

Well, you could. But here are some differences.

If you put this in a stored procedure, then you won’t be able to use the stored procedure in a SELECT statement.

If you put this in a stored procedure, you are likely to put all of the WHERE clauses you need in the stored procedure. This will grow to the point where you won’t have code that is easy to maintain.

In short, you should use every opportunity to make your SQL as modular as possible. Just like you should be doing in any other language you use to create your applications. Just because we’ve changed languages, doesn’t mean that all of the best practices you should have learned have gone flying out the window.

FUNCTIONS

When I started out writing this book, I wasn't going to include functions. In fact, when I wrote version 1 of this book, there was a lot I left out. But at this point, I've written about everything else, so I might as well include functions.

The first thing I'll say about functions is this:

You'll probably never need them.

That's right. Most of what you are going to want to do is use a stored procedure and return a row. The only reason you would ever write a function is if you needed to modify individual strings, numbers or other field types.

I've been programming for 28 years and in that 28 years I've only met 2 database guys who I would say actually knew what they were doing with SQL. Of the two, only one used functions regularly. The reason he did this was because he placed all of his business logic in his database. If you believe the database is where your business logic belongs, I guess you have a reason to put functions in your database.

But most senior level programmers I know believe that kind of logic doesn't belong in the database. The database is for data. Business rules belong outside of the database.

But, let’s just assume for a while that you’ve found an exception and you really need some function that isn’t already available in the SQL language. How would you create, modify, and delete a function?

Before we answer that question, the first question you might want to ask is, “What makes a function different from a stored procedure?”

Here are some key points:

- Functions only return one value.
- Functions only have input parameters. Stored Procedures allow for output parameters (and in/out parameters)
- Functions don’t act on the database.
- Functions can’t return XML data type
- Functions don’t support exception handling

Scalar Functions

The basic syntax for creating a function would look like this:

```
CREATE FUNCTION functionName
(
 /* parameter list */
)
RETURNS /* return type */
AS
BEGIN
   /* computation */
   RETURN (SELECT /* return value */);
END
```

A couple of things to note.

First, the parameter list is going to look just like the parameter list for a stored procedure, with the exceptions I've already mentioned above.

Second, you have to tell the function what you'll be returning. This would look like the data type of the parameters.

Third, you need a Begin and End statement to specify where the function starts and stops. This lets you have multiple returns.

So, a fullName function might look like this:

```
CREATE FUNCTION fullName
(
 @firstName varchar(50),
 @lastName varchar(50)
)
RETURNS varchar(101)
AS
BEGIN
    DECLARE @fullName as varchar(101);
    SET @fullName = @firstName + ' ' +
@lastName;
    RETURN (SELECT @fullName);
END
```

And you can use this function just like any other function in your SQL select statements or anywhere else you might use a function.

Scalar functions are probably the easiest to understand because they work a lot like the functions we use every day in our regular programming work.

But there is another kind of function called a Table-valued Function (TVF)

Table-valued Functions

The main difference between a scalar function and a table-valued function is how they are used. Scalar functions get used much like a sub-select would. That is, they get called for each row and for each place they are placed in your code.

On the other-hand, table-valued functions are computed all at once and return a table which you can JOIN to. And, just like JOINs they end up being a lot more efficient.

What makes a table-valued function a table-valued function is that it returns a TABLE instead of s scalar.

So, to create a table-valued function, the code will look a lot like what I've already shown you with the exception that the RETURNS is going to be. There is also no need for the BEGIN and END bracketing.

```
CREATE FUNCTION functionName
(
 /* parameter list */
)
RETURNS TABLE
AS
   RETURN (SELECT * FROM SomeTable …);
```

This is a single-statement table-valued function. What makes it single-statement is that it is essentially a select statement that returns a table. This is why it doesn't need the BEGIN and END bracketing.

But what if you want to create a function that has multiple statements in it and it returns a TABLE. It is still a table-valued function, but it is a muti-statement table-valued function.

Here are the key differences.

- You will need to delimit the BEGIN and END of your function like you did with scalar functions.
- You will need to declare a variable that will be used for the return value.

```
CREATE FUNCTION functionName
(
 /* parameter list */
)
RETURNS @returnTable TABLE
AS
BEGIN
   /* other code here as needed */
   INSERT INTO @returnTable ...
   RETURN;
END
```

The key is that you will insert records into your return table and that will become the data you will JOIN to.

There is a slight down side to table-valued functions and that is for functions of any complexity, you'll need to pay attention to optimizations. This goes double for multi-statement table-valued functions because there may be other stuff going on that the SQL optimizer can't automatically optimize.

TRIGGERS

Triggers are essentially stored procedures that are attached to specific events. In the early days of SQL, the events they were attached to were rather limited. But today, we can attach events to so much more that they are worth learning as a programmer because they might just solve some problems you would struggle to solve any other way.

The basic syntax looks like this:

```
CREATE TRIGGER triggerName
ON tableOrView
FOR SomeTypeOfEvent
AS
  /* Your stored procedure code here */
```

There are other qualifiers available, but we are going to stick with the basics. If you are interested, you can use your favorite search engine to find out more.

What do we have here. The obvious first step is to give your trigger a name. Where things get interesting is that you can create a trigger on a table or a view when any of the following events occur:

- INSERT

- UPDATE
- DELETE

But there are a few wrinkles you need to know about before you go off making assumptions about how these work that just aren't true.

For example, let's say that you do a bulk insert and insert 20 records. Will your INSERT trigger fire 20 times or once?

You might be inclined to think it would fire once. But think about it. How efficient would that be? No, it makes more sense for it to fire once and let you deal with the details for each record that was created individually.

Next, you might think that the data gets passed to you. But that would only work if the stored procedure fired for each record. Since it doesn't, the more efficient way of dealing with the data is by putting it in a table that you can get a hold of once you are in the appropriate trigger. There are two special tables that are created right before your trigger is executed. INSERTED holds all of the records that were inserted and DELETED holds the records that were deleted. Of course. "But," you might ask, "what about updates?" A record that is updated will appear in both the DELETED and INSERTED tables. Both of these tables have the same structure as the table, or view, the trigger was created on.

So, now you have the basics. You can create an INSERT trigger using the syntax:

```
CREATE TRIGGER triggerName
ON tableOrView
FOR INSERT
```

```
AS
  /* Your stored procedure code here */
```

If you want to use the same trigger for multiple events, you can comma separate the events.

```
CREATE TRIGGER triggerName
ON tableOrView
FOR INSERT, UPDATE, DELETE
AS
   /* Your code here */
```

Now, I mentioned that you can create triggers on Views. That might seem pretty odd. Most of the time, the reason we created a View was to join two different tables together. So, how would an update, insert or delete work on a view?

Well, for this situation, we replace the FOR with INSTEAD OF. What we are saying is, when someone tries to insert, update, or delete the table or view this trigger is attached to, don't apply the operation to the table. Instead, run this stored procedure. One would hope you would write the stored procedure part in such a way that would be smart enough to update the tables behind the view so that it looks like you did the appropriate update operation.

Triggers can be used for more than just tables and views, but this is the most common scenario. They can now also be used to execute code when someone logs into the database, or when rights are modified. But if you are just getting started, I would spend most of my time learning how they apply to table and views because that is where you'll have the most use for them as a programmer.

Testing

While testing has been a topic that has gained quite a bit of popularity over the years, and one could argue has become something of a religion, with people being for it AND against it. The topic of testing databases has just recently started to gain traction in the database community. This is mostly because testing is harder in the database world than in any other environment.

What I want to primarily focus on in this chapter is unit testing. And for unit testing to work in any environment, what we need to do is create small units of work that are testable.

The reason testing is hard in any environment is that the code is not modular enough.

Second, I find that most people try to test their database against a real database, or a database that has real-*ish* data in it.

Using real data is great if you are doing performance testing. But it stinks for unit testing because you really have no idea if the data you got back is the data you SHOULD have gotten back.

What we want, instead, is known data that assist in helping us determine if a stored procedure, or table, or view, is working the way we would expect it to work.

So, for any given test, what we want to do is create a separate database that has only the tables in it that the particular bit of functionality we are testing needs.

And just like in our C#, Java, VB.NET, or JavaScript code. We want to isolate, or mock out, our dependencies.

This would mean that if you are testing a stored procedure that is using a view, you'll want to replace the view with a table that has known data.

Or maybe you are testing a stored procedure that uses some other stored procedure. The other stored procedure should be created in such a way that it returns known data.

In this way, we are only testing one bit of functionality at a time.

This will mean that instead of creating one great big monolithic database with all of your tables, stored procedures, and views in it, what you'll be doing is creating a setup routine that sets everything up for the thing you are planning on testing. Then it will run the test. It will then verify that it got back what it expected to get back. And finally, it will delete everything so the database you are testing with is empty prior to starting the next test.

By writing your test in this way, testing becomes easy and you end up with a high level of confidence that your code works.

But unit testing isn't the only thing you should be testing. I'm sure you've heard stories of programmers who put together database systems that ran fine during development only to discover that once the system went into production, it couldn't keep up.

Just this past week I heard a story from a fellow programmer who told me of an existing internal system used by a very popular company. They decided to be cleaver and created a name/value pair system instead of putting it all in one row. It should be no surprise to anyone that once this went into production it couldn't hold up in real day-to-day use. It is SO slow that they've put a process in place to only store one month's worth of data, they only return a maximum of 500 rows, and they are using Lucene to index and search the data.

They obviously have a problem. Since the group I am in needs a faster way of retrieving the data, we are going to shadow the data into an optimized database.

But this could have been avoided if performance had been part of the criteria. As the programmer I was discussing this with said, "All they would have had to do is throw in a million fake 'rows' and they would have known right away they had a problem before they built out the rest of the system."

I'm not saying you need to optimize the guts out of a system, but anything that takes more than a few seconds to return a couple thousand rows, definitely has an issue.

Version Control

As I've already mentioned, one of the other areas that programmers have paid quite a bit of attention to over the years and the database world has only recently started thinking about is how to version the database.

One of the things we don't want to do is to version the entire database with all of its data. Yes, if we have scripts that create lookup tables, we probably want to put those into our version control system. But generally all we want to version is the changes to our scripts that create the tables, stored procedures and other artifacts of getting a database in place for our application.

But we also don't want to produce a CREATE script for each element in our database system.

The best way I've seen to version database scripts is to create a script that creates the entire first cut of the database and put that in a file named 00.00.0000.sql.

Every time you make a change to the database, you create a script that scripts the change and you create a new file, 00.00.0001.sql, 00.00.0002, etc.

To create a new database that is today's version of the database, you would run each script in order.

If you are cleaver, you will store the version number of the script in the database in its own table so you know exactly what version the database corresponds to.

You may think this to be an onerous task, but there are several tools available now that basically write the script for you. Some that come with tools from Microsoft and Third Party tools as well. It isn't that hard and may just save your neck.

You might wonder what the 00.00. part of the number scheme is for. Well, eventually you are going to release the code to production. At that point, the version number becomes 01.00.0000 and then there may be the occasional bug fix which will produce version 01.01.0000. The last set of digits are for incremental development.

Resources

If you liked this book, I'd really appreciate it if you took the time to write a review on Amazon.

Did you find mistakes along the way? Did I not explain something enough? Let me know. The great thing about how books get published today is that they can be updated much more quickly than in the past.

The best way to contact me is through email at davembush@dmbcllc.com.

Thanks for joining me on this SQL journey. When I started out, typically all I could find was great big 500 page books that told me all I could possibly want to know about SQL. Frankly, I got lost. Where was the "Here's the minimum you need to know to get going." Book?

Well here it is.

But if you want more, here are some links to other places you might want to take a look at.

Of course, there is always my blog at **http://blog.dmbcllc.com** where I write about a variety of topics related to the programming work I'm doing.

And I'm a huge fan of the guys at PluralSight which you can get to by following this link: **http://blog.dmbcllc.com/pluralsight**

By the way, if you are a .NET Programmer, you might be interested in "NUnit 3.0 with Visual Studio 2015" which you can find at:

http://blog.dmbcllc.com/nunit3WithVS2015