CHAPTER 4

Tables

Imagine buying a new computer. You bring home the box, full of excitement. You open the box—and it's empty! You have a box, but it's not much use, is it? Databases are exactly the same. If you have a database without tables, it serves no purpose. Tables are the data containers. Everything you might want to store in a database—customer information, order details, your comic collection—has to go into a table. Virtually everything you build in your database will rely on data from tables.

I think it's pretty clear that tables are quite important, so we'd better figure out how they work and how we can create them!

Table Basics

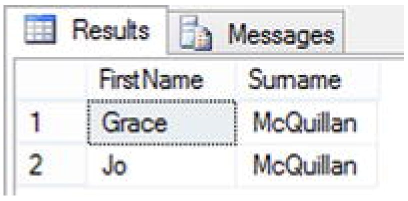
Tables in databases do not consist of four legs and a tabletop. Now that would be interesting! No, database tables consist of two basic elements:

- Columns
- Rows

Together, these two items form a gridlike structure. The columns define the pieces of data we want to store, and the rows hold the values for those columns. A row is just a collection of column values. Here's a quick example:

- We have a table called `Customers`
- `Customers` has two columns: `FirstName` and `Surname`
- We add a new row: a `FirstName` of `Grace`, and a `Surname` of `McQuillan`
- We add a second row: a `FirstName` of `Jo`, and a `Surname` of `McQuillan`

The table now contains two rows of data, one for Grace McQuillan, and one for Jo McQuillan—just like Figure 4-1, in fact.



	FirstName	Surname
1	Grace	McQuillan
2	Jo	McQuillan

Figure 4-1. *Two rows of data*

Just having a list of names in a table isn't much good to us. For example, what would happen if a second Grace McQuillan were to be added? How could we tell them apart? How could we link these names to other pieces of data, such as orders made by a customer? How can we prevent duplicate records from being added? The answers to all of these questions and more will become clear as we proceed. Let's have a look at our first table script.

A Starting Point

Use the `CREATE TABLE` SQL statement to create a table in SQL Server. Take a look at this script.

```
USE AddressBook;

CREATE TABLE dbo.Contacts
(
ContactId INT,
FirstName VARCHAR(40),
LastName VARCHAR(40),
DateOfBirth DATE,
PhoneNumbers VARCHAR(200),
AllowContactByPhone BIT,
FirstAddress VARCHAR(200),
SecondAddress VARCHAR(200),
RoleId INT,
RoleTitle VARCHAR(200),
Notes1 VARCHAR(200),
Notes2 VARCHAR(200),
DrivingLicenseNumber VARCHAR(40),
PassportNumber VARCHAR(40),
ContactVerified BIT,
CreatedDate DATETIME
);

GO
```

First we meet the `USE` statement we saw in Chapter 3, but this time we are saying `USE AddressBook`. So whatever we run as part of this batch of T-SQL statements will be executed within the **AddressBook** database. Then we execute a `CREATE TABLE` statement. This is the only real piece of code in this script. It will create a table called `Contacts` with the appropriate columns. Then we finish up with our now ubiquitous `GO` statement.

WHAT IS DBO FOR?

You may have noticed we didn't call the table `Contacts`—rather, we called it `dbo.Contacts`. `dbo` is the default schema. A *schema* is a container that allows you to group together objects in a folder-like way. If you had a set of reporting objects (tables, functions, stored procedures) you could put them all into a schema called `Reporting`.

Schemas are very useful as they allow you to define security permissions for the entire set of objects, rather than one object at a time. They are also great for logically separating your database objects.

Unless you explicitly provide a schema name, all objects you create will be added to the `dbo` schema. The `dbo.` in the preceding example is optional, but it is good practice to be explicit whenever you can.

Schemas are discussed more in Chapter 20.

Save this script to `c:\temp\sqlbasics\apply\02 - Create Contacts Table.sql`. Once saved, run the statement by pressing F5 and the table will be created. To view the table, expand the **Databases** node in the Object Explorer, then **AddressBook**, then **Tables**. As Figure 4-2 shows, you'll see **dbo.Contacts** in the list (you may need to right-click the **Tables** node and choose the **Refresh** option).

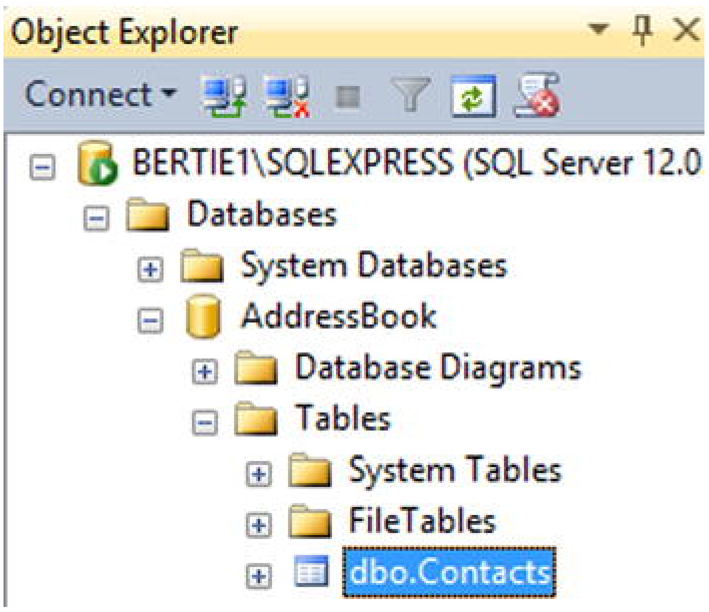
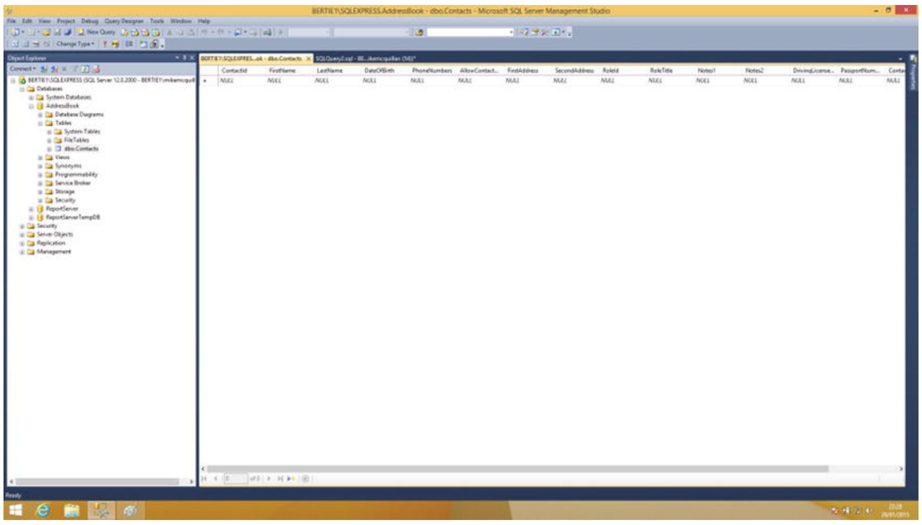


Figure 4-2*. The `Contacts` table in Object Explorer*

Now right-click `dbo.Contacts` and choose **Edit Top 200 Rows**. The table opens up, awaiting data entry (see Figure 4-3).


BERTIE1\SQLEXPRESS.AddressBook - dbo.Contacts - Microsoft SQL Server Management Studio
BERTIE1\SQLEXPRES...ok - dbo.Contacts
SQLQuery2.sql - BE...ikemcquillan (54))*
Object Explorer

	ContactId	FirstName	LastName	DateOfBirth	PhoneNumbers	AllowContact...	FirstAddress	SecondAddress	RoleId	RoleTitle	Notes1	Notes2	DrivingLicense...	PassportNum...	Conta
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Figure 4-3*. `Contacts` table, awaiting data entry*

CHANGING THE NUMBER OF ROWS

You can modify how many rows SSMS will return in the **Edit Top 200 Rows** context menu option. Go to **Tools ➤ Options**, and click **SQL Server Object Explorer** on the left. You can change the values for the appropriate options here—you may choose to display the top 500 rows, for example.

Our first table now exists. You could have used SSMS to create the table had you wished; simply right-click the **Tables** node and choose the **Table** option. This will display the table editor, which is shown in Figure 4-4.

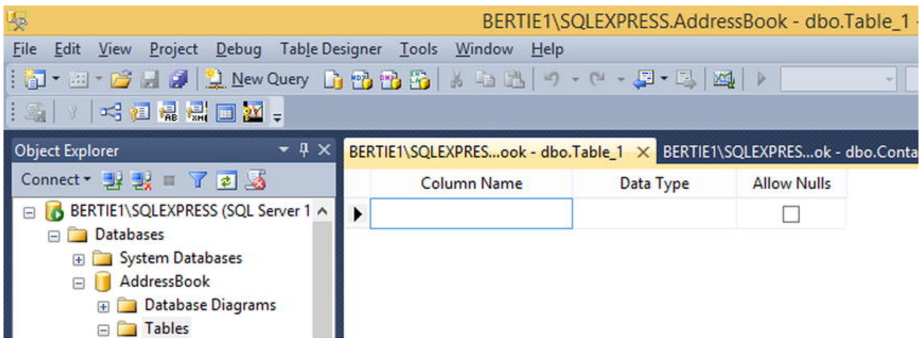
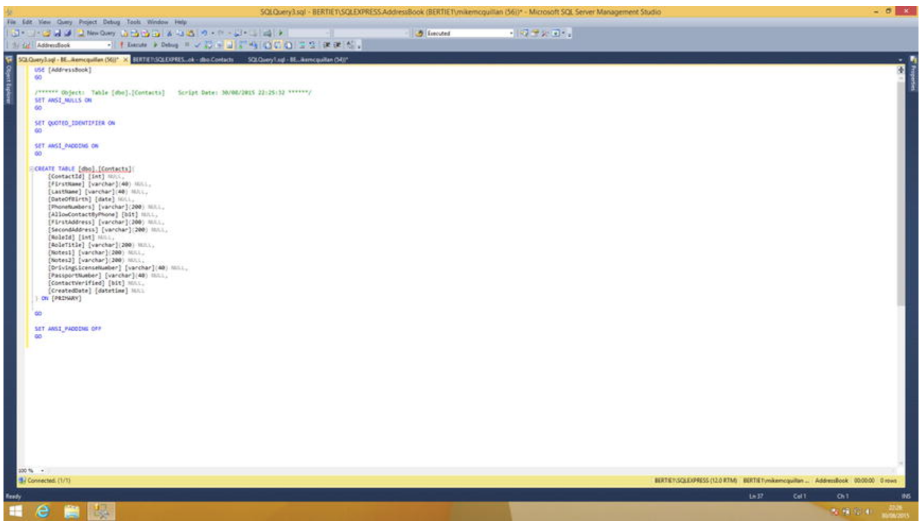


Figure 4-4*. Creating a new table using the table editor*

Of course, using the table editor means our code is not reusable. However, you could create the table using SSMS and script it, as we did with our database in the last chapter. Right-click the `Contacts` table, select **Script Table as**, then **CREATE To**, and finally click **New Query Editor Window**. As Figure 4-5 shows, the `CREATE TABLE` statement will appear, again with more detail than we originally provided (remember, `CREATE DATABASE` did a similar thing).



```
USE [AddressBook]
GO

/****** Object:  Table [dbo].[Contacts]    Script Date: 30/08/2015 22:25:32 ******/
SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO

SET ANSI_PADDING ON
GO

CREATE TABLE [dbo].[Contacts](
    [ContactId] [int] NULL,
    [FirstName] [varchar](40) NULL,
    [LastName] [varchar](40) NULL,
    [DateOfBirth] [date] NULL,
    [PhoneNumbers] [varchar](200) NULL,
    [AllowContactByPhone] [bit] NULL,
    [FirstAddress] [varchar](200) NULL,
    [SecondAddress] [varchar](200) NULL,
    [RoleId] [int] NULL,
    [RoleTitle] [varchar](200) NULL,
    [Notes1] [varchar](200) NULL,
    [Notes2] [varchar](200) NULL,
    [DrivingLicenseNumber] [varchar](40) NULL,
    [PassportNumber] [varchar](40) NULL,
    [ContactVerified] [bit] NULL,
    [CreatedDate] [datetime] NULL
) ON [PRIMARY]

GO

SET ANSI_PADDING OFF
GO
```

Figure 4-5. Generating a `CREATE TABLE` statement

WHAT'S ALL THAT EXTRA STUFF?

You're probably wondering what `SET ANSI_NULLS ON`, `SET QUOTED_IDENTIFIER ON`, the square brackets and so on in Figure 4-5 are for.

`SET ANSI_NULLS ON` dictates how we find records with NULL values (you'll find out about NULLs in Chapter 7).

`SET QUOTED_IDENTIFIER ON` determines whether double quotes can be used in SQL statements (so `Notes1` works the same as `Notes2`).

`SET ANSI_PADDING ON` tells SQL Server how to store text values, and whether spaces at the end of text values should be kept or not.

The square brackets around the various names in the statement in Figure 4-5 are completely optional. They are not needed for any of the preceding terms as they are all fairly simple names. They are required when a name has a special character in it, like a space or parentheses. So the name `Role Title` would cause an issue, but `[Role Title]` would work fine. Special characters should be avoided.

Last, each column declaration has the word `NULL` after it. This means `NULL` values can be stored in the columns. You'll see more of this in Chapter 7, too.

Okay, you have the extreme basics of `CREATE TABLE` down. Now we need to take a look at what creating a table actually involves.

Naming Tables

It is important that you name your tables accurately. For example, the test table we just created is called `Contacts`, because it is going to house contact information. Which leads us to ask:

To s or not to s, that is the question.

Some people may be reading this and thinking, "Never put a letter S on the end of a table name! Don't pluralize it, you fool!" And you could argue they are right. But other people will be thinking, "Yes, pluralize away." And you could argue they are right, too. The simple answer here is you can choose to pluralize or not, as long as you are consistent. One way isn't better than the other.

Some people like: `Contacts`.

Other people like: `Contact`.

The choice is yours. Personally, I always pluralize. My only rule against this is I don't pluralize when somethi ends in a Y. So if I had a `Responsibility` table, I would change the name to something like

ResponsibilityItems, not Responsibilities.

Table names can begin with a letter, an underscore, an @ sign, or a # sign. Unless you are creating a temporary table (see the next section), I recommend you always begin with a letter. Keep the names simple and avoid special characters where possible.

Types Of Table

SQL Server supports a few different table types. Let's see what we have in Table **4-1**.

Table 4-1. *SQL Server Table Types*

Table Type	Description
Heap Table	A normal, permanent table, used to store rows of data. The Contacts table we just created is a heap table. This is a table without a clustered index (see Chapter 14).
Clustered Table	A normal, permanent table, but with a clustered index (again, see Chapter 14!).
Local Temporary Table	Temporary tables always begin with a # symbol. These are tables that only exist inside a particular block of code. They are temporarily created inside **tempdb** and removed once the code has finished (or the window in which the code is executing is closed).
Global Temporary Table	This begins with ##. It is available to all users of the system until it is destroyed. It must be destroyed manually, unless the system is restarted. Whenever SQL Server restarts it rebuilds **tempdb**, so any temporary tables in there would be removed.
FileTable	These were new in SQL Server 2012. Do you remember our brief discussion on FILESTREAM in Chapter 2? FileTables store FILESTREAM data.

We'll be dealing with clustered tables for the majority of the book.

Columns and Table Data Types

Look at these two lines from our earlier CREATE TABLE statement:

```
ContactId INT,
FirstName VARCHAR(40),
```

These two lines are column declarations. The first defines a ContactId column of type INT (integer), which means the column can store a number. The second, FirstName, is of type VARCHAR(40). This means the value can be any combination of characters, up to a maximum of 40 characters. This is known as a *string* in many programming languages.

The column names make it clear what the purpose of each column is, and this is the thing to remember when naming your columns: always make the meaning clear. Also, avoid special characters if you can; otherwise you may find yourself having to use square brackets. One other point: Always explicitly name columns, especially identifiers. Don't use generic names like `ID` or `Name`; use `ContactId` or `ContactName` instead. The reason for this will become clear later when we look at relationships. In some special circumstances there may be a valid reason for adding a column called `ID`, but this is outside the scope of this book.

T-SQL offers many different data types. The table in Appendix A lists them all, but in general you'll find yourself using `INT/BIGINT`, `CHAR/VARCHAR/NVARCHAR`, `BIT`, and the `DATETIME` options more than anything else. We'll have a quick glance at the data types we are going to use in our examples.

- `INT`: Numeric values. You can store any whole number in an `INT` column, from about –2 billion to 2 billion (exact numbers are in Appendix A!). `BIGINT` is similar but it can store much larger numbers.
- `VARCHAR`: This stands for variable character. You have to define a number to specify the length of this (between 1 and 8,000) in bytes. You can have strings longer than 8,000 by specifying `VARCHAR(MAX)` (this allows values up to 2GB in length). You declare a `VARCHAR` of 25 characters, for example, by specifying it as `VARCHAR(25)`.

  Only the actual data stored in the column takes up room; so if you did have a `VARCHAR(25)` and the name `Dolly` was entered, only five characters would be stored. This is different from `CHAR(25)`, which would hold 25 characters (`Dolly` and 20 spaces). `NVARCHAR(25)` would also use five characters for `Dolly`, but it needs double the space to store each character, as the value is stored in Unicode format.
- `BIT`: This denotes if something is true or false. The value is stored as `0` or `1` (or `NULL`, if allowed). In a `Contacts` table, for example, you might have a `BIT` column for `ContactByPhoneAllowed`.

THREE-STATE LOGIC

`BIT` columns support something called three-state logic. Normally, a `BIT` column only allows two values: `TRUE` or `FALSE`. But if you allow `NULL` values to be stored in the column, too, you can store three different responses. This capability can often come in handy.

- `DATETIME`: Stores a date and time combination. This is the most commonly used date/time field you'll come across in SQL Server databases, as it was the main date/time storage mechanism until separate `Date` and `Time` data types appeared in SQL Server 2012.

UNICODE FORMAT

Unicode is a standardized method that ensures consistency across languages used on computers. It supports multiple characters across many languages, including letters and scripts (e.g., it supports Arabic,).

If your database needs to support multiple languages, you should use Unicode data types like `NCHAR`, `NVARCHAR`, and `NTEXT`. You can use the standard non-Unicode types if you don't have this requirement.

Be careful when deciding how long your string-based columns are going to be. It's fairly easy to make a column bigger, but this has ramifications on other parts of your systems, as we'll see.

Primary Keys

One thing our `Contacts` table doesn't have is a `Primary Key`. This is a column (or combination of columns) that can be used to uniquely identify a row. Let's look at an example: me and my dad. Some years ago, we had the same name and address; only our dates of birth differed. As a database record, it would be quite difficult to uniquely identify either of us—you'd need to use name, address, and date of birth. This isn't a good solution. As we develop a database, we add more tables to it and link those tables together using foreign keys, which we'll look at in the next section. It's simpler to use one column in foreign key relationships.

Another thing we want to do is keep primary keys concise. If they are small, this allows us to pull records back from the database faster than if we were using a combination of name, address, and so on. It is standard practice to use numeric fields, like `INT` or `BIGINT`, as primary keys. We added a `ContactId` column to our Contacts table. That column exists to serve one purpose only: to uniquely identify the row. It is there to act a primary key.

Let's modify our `CREATE TABLE` T-SQL statement to turn `ContactId` into a primary key column. First, we'll need to drop the current version of the table. Do you remember the test we added in Chapter 3 to check if a database already existed? We used it to look at `sys.databases`. Similarly, we'll look at `sys.tables` to check if our table exists. If it does, we'll drop it, then our code will execute and recreate it.

Open `c:\temp\sqlbasics\apply\02 - Create Contacts Table.sql` and change it so it looks like this:

```
USE AddressBook;

IF EXISTS (SELECT 1 FROM sys.tables WHERE [Name] = 'Contacts')
BEGIN
DROP TABLE dbo.Contacts;
END;

CREATE TABLE dbo.Contacts
(
ContactId INT,
FirstName VARCHAR(40),
LastName VARCHAR(40),
DateOfBirth DATE,
PhoneNumbers VARCHAR(200),
AllowContactByPhone BIT,
FirstAddress VARCHAR(200),
SecondAddress VARCHAR(200),
RoleId INT,
RoleTitle VARCHAR(200),
Notes1 VARCHAR(200),
Notes2 VARCHAR(200),
DrivingLicenseNumber VARCHAR(40),
PassportNumber VARCHAR(40),
ContactVerified BIT,
CreatedDate DATETIME
);

GO
```

Our `CREATE TABLE` statement remains exactly as it was, but we've added an `IF EXISTS` check at the top. We are checking if the table exists by looking for its name in `sys.tables`. If the table does exist, we drop it. Then our `CREATE TABLE` statement runs without any issues.

Note that I've wrapped the `[Name]` column in square brackets. Why is this? Didn't I say they were only needed for names with special characters in them, like a space? Indeed I did. But I also recommend using them when column or object names are *reserved words*: words SQL Server uses as part of T-SQL, for example. `Name` happens to be a SQL Server keyword, so I wrapped it in square brackets to explicitly tell SQL Server I am using it as a column name.

RESERVED WORDS

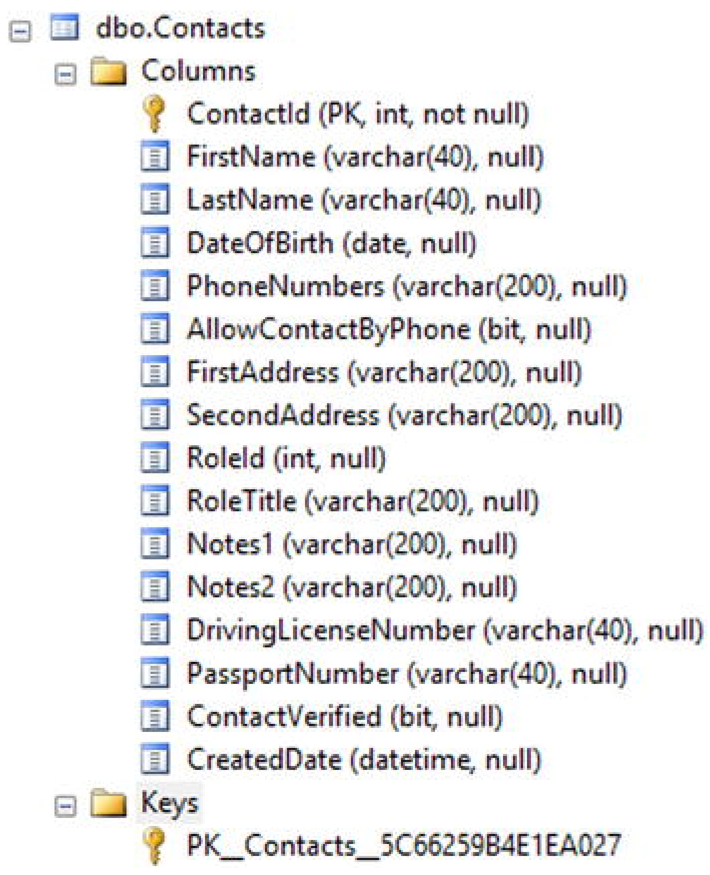
SQL Server reserves a set of words that have a special meaning within SQL Server—`SELECT`, for example. Most reserved words act as statements in the T-SQL language, like `EXISTS`.

I recommend avoiding the use of reserved words wherever possible to avoid confusion. If you have your heart set on a particular word, just remember to wrap the object name (e.g., a table or column) in square brackets, like we did with `[Name]`.

Now that we know we can drop and recreate the table at will, we can add our primary key to it. There are two ways to do this. The first one is to change the ContactId line so it looks like this:

```
ContactId INT PRIMARY KEY,
```

Run the script again. In Object Explorer, right-click **Tables** and then **Refresh**. Expand **Contacts**, then expand **Columns** and **Keys**. You'll see something like Figure 4-6.



dbo.Contacts
Columns
ContactId (PK, int, not null)
FirstName (varchar(40), null)
LastName (varchar(40), null)
DateOfBirth (date, null)
PhoneNumbers (varchar(200), null)
AllowContactByPhone (bit, null)
FirstAddress (varchar(200), null)
SecondAddress (varchar(200), null)
RoleId (int, null)
RoleTitle (varchar(200), null)
Notes1 (varchar(200), null)
Notes2 (varchar(200), null)
DrivingLicenseNumber (varchar(40), null)
PassportNumber (varchar(40), null)
ContactVerified (bit, null)
CreatedDate (datetime, null)
Keys
PK__Contacts__5C66259B4E1EA027

Figure 4-6. *A dynamically named primary key*

Two things of note here. First, the `ContactId` icon has changed to a small key, and it says `PK` next to it. This tells us the column is acting as a primary key. Second, we have an entry under **Keys**. This also has a key icon and begins with `PK_Contacts`, for "primary key." The random letters and numbers have been added by SQL Server to create a unique name.

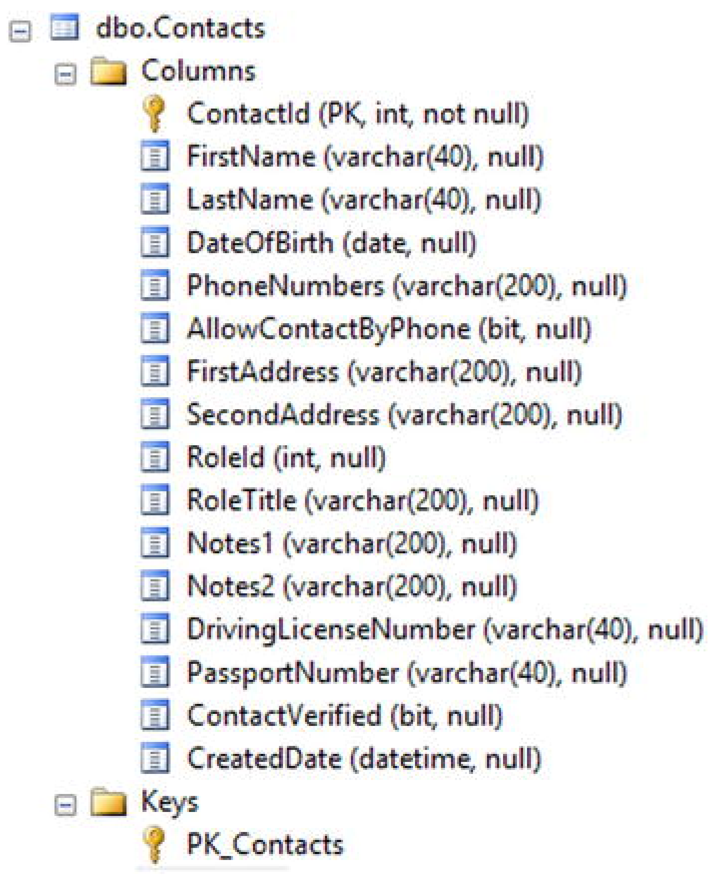
Now, I don't like this method of creating primary keys. I don't want a load of random letters and numbers at the end of my names; I prefer to exercise more control by explicitly naming my objects. Remove `PRIMARY KEY` from the ContactId line, so it looks like this again:

```
ContactId INT,
```

Find the `CreatedDate` line, and add a comma at the end of it. Then, type in the second line in the following script:

```
CreatedDate DATETIME,
CONSTRAINT PK_Contacts PRIMARY KEY (ContactId)
```

Run your script again, refresh **Contacts** in Object Explorer, and expand **Columns** and **Keys**. Now you will see the good primary key name, just like in Figure 4-7:



```
dbo.Contacts
  Columns
    ContactId (PK, int, not null)
    FirstName (varchar(40), null)
    LastName (varchar(40), null)
    DateOfBirth (date, null)
    PhoneNumbers (varchar(200), null)
    AllowContactByPhone (bit, null)
    FirstAddress (varchar(200), null)
    SecondAddress (varchar(200), null)
    RoleId (int, null)
    RoleTitle (varchar(200), null)
    Notes1 (varchar(200), null)
    Notes2 (varchar(200), null)
    DrivingLicenseNumber (varchar(40), null)
    PassportNumber (varchar(40), null)
    ContactVerified (bit, null)
    CreatedDate (datetime, null)
  Keys
    PK_Contacts
```

Figure 4-7. *A primary key named by the developer*

Much better! We now have a concise, to-the-point primary key name. In truth, you can call the primary key anything you want, but using `PK` is good practice—it tells developers the key's purpose.

Clustered and Nonclustered Keys

Primary keys, whether they use one or more columns (multiple-column primary keys are known as *compound keys*), can be either clustered or nonclustered.

- A *clustered* primary key determines how data are stored on the hard disk
- A *nonclustered* primary key supports fast querying of data, but doesn't dictate how the data are stored on disk

We'll see these terms again when we look at indexes in Chapter 14, where we'll walk through an example of how each of these items can affect how your table stores data. Indeed, a primary key is nothing more than a special type of index. By default, a clustered primary key will be created if you do not explicitly state what type of primary key you want to create.

Note You can only have one clustered index per table. This is usually the primary key, but it doesn't have to be, as we've just seen. However, you can have lots of nonclustered indexes on a table, and SQL Server will have you covered.

To explicitly create a nonclustered primary key, change our new `CONSTRAINT` line to include the `NONCLUSTERED` keyword:

```
CONSTRAINT pk_Contacts PRIMARY KEY NONCLUSTERED (ContactId)
```

Run this and your table will be recreated. You should use `ContactId` as a clustered column, as you'll be usi
it to identify lots of pieces of data, and having it in the correct order will speed up queries. So change that lin
to use the `CLUSTERED` keyword instead:

```
CONSTRAINT pk_Contacts PRIMARY KEY CLUSTERED (ContactId)
```

Run this and save the script. Strictly speaking, we don't need to provide the `CLUSTERED` keyword (this is the default), but it is always better to be explicit—it makes your code easier to understand. Think about the next person!

Foreign Keys and Relationships

I briefly mentioned foreign keys earlier. We use these to link two tables together. One table is a parent and the other is a child. We won't create any foreign keys in this chapter, but it's a good idea to take a look at what they are, to steel yourself for the challenges ahead!

You might think you can create one big table to hold all your data, which is what I did earlier with the `Contacts` table. This is a very bad approach and will make things difficult for you when you need to generate reports or find individual pieces of data. Over the rest of this chapter and the next, you'll see how logically separating data into different tables makes sense.

Say we have two tables: `Contacts` and `ContactPhoneNumbers`. One contact could have many phone numbers. From this simple sentence we can derive that the `Contacts` table (the "one") is the parent, while `ContactPhoneNumbers` (the "many") represents the children. They can have a few types of relationship:

- One to one
- One to many, or many to one
- Many to many

You'll come across lots and lots of one-to-many relationships, and quite a few many-to-manys, too, but not so much on the one-to-one front. Not that it isn't perfectly valid; it just isn't used as much as the other two. Here's what each of these relationships is used for, along with an example for each.

- **One to one**: Normally used to split a large table into several different subtables. Assume a `Contacts` table like the one we saw earlier. Now assume we need to store lots of nationality information. Rather than add those columns to the `Contacts` table, we could create a `ContactVerificationDetails` table and link it to the `Contacts` table. You do this by specifying the same primary key for both tables. Both tables would have `ContactId` as the primary key, which would be used to link the tables together. You can find the primary key in the table by looking for a column name with a key next to it.

  You can see this layout in Figure 4-8. The "one" sides of the relationship are represented by keys at each end of the adjoining line.

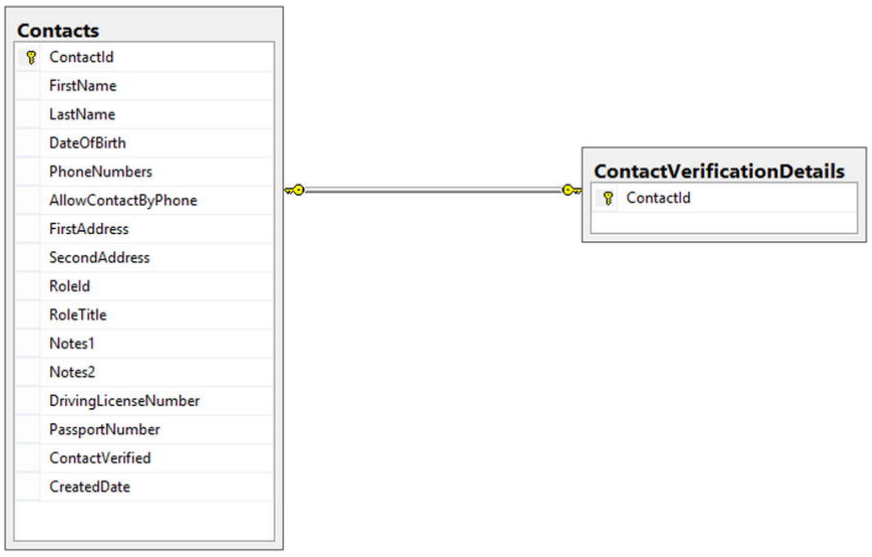


Figure 4-8. One-to-one relationship diagram

- **One to many**: This is where a parent record in one table "owns" child records in another table. The records in the "many" table could not exist without the parent. An example here is a record in `Contacts` as the parent, with a `ContactPhoneNumbers` table containing the child records. Phone numbers are of no use unless we know to whom they belong.

We've set this up by defining a different primary key on each table, but including the `ContactId` column —the parent record's primary key—in both tables. We then define a relationship between those columns to link the tables together.

Figure 4-9 uses the key we've already seen for the "one" side of the relationship (`Contacts`), and a figure eight for the "many" side (`ContactPhoneNumbers`).

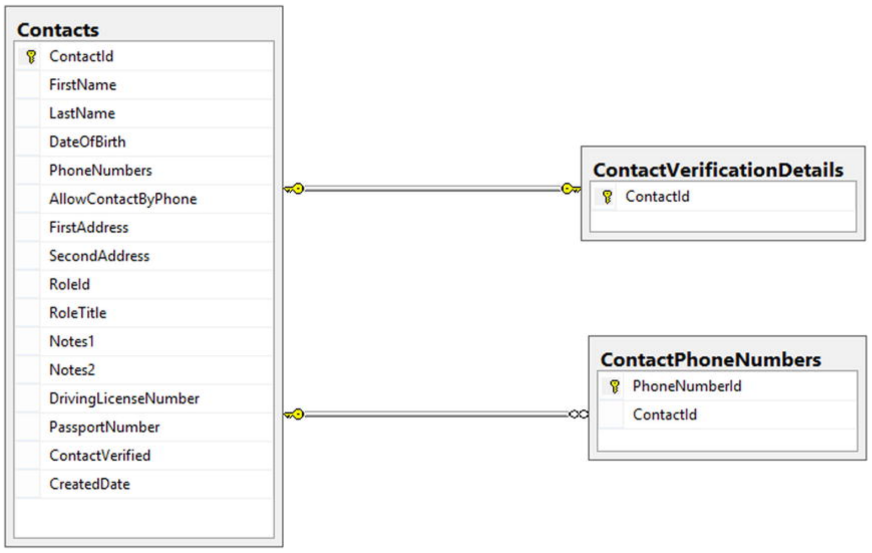


Figure 4-9. One to many relationship diagram

- **Many to many**: This is the most complicated, because it requires three tables to implement it correctly. What if `Contacts` were assigned to `Roles`? The roles available would stay the same, but they could be assigned to different contacts as they move between different jobs. Indeed, one role may be assigned to multiple contacts—for example, if we had twenty SQL Developers. What we are saying here is one contact can have one or more roles, while one role can be assigned to one or more contacts. We have duality. We cannot link the tables directly, as doing this would necessitate having either `Contacts` or `Roles` as the parent—not good, as in this case they are both parents.

  The solution is to add a table in the middle, holding both the `Contacts` and `Roles` primary key values as a unique combination. This combination will form a *compound key*: a primary key consisting of more than one column (two in this case). Figure 4-10 shows this implementation.

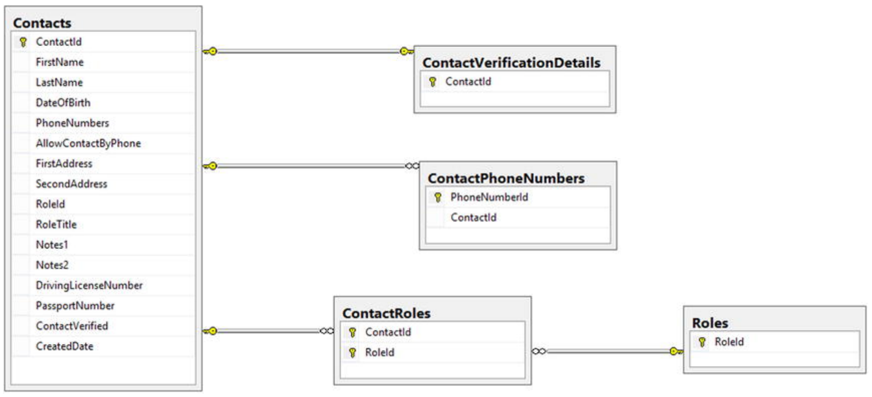


Figure 4-10. Many-to-many relationship diagram

Here, we have a one-to-many relationship heading from `Contacts` to `ContactRoles`, which is the table facilitating the many-to-many relationship. You can see a key in this table against both `Contact` and `RoleId`; this shows that both columns together are acting as the primary key.

You can see a similar relationship starting at `Roles` and finishing at `ContactRoles`, too. Note that both figure eights are linked to `ContactRoles`, thus defining it as the table holding the many-to-many relationship records.

We're now going to build some tables in our `Contacts` database and link them together. As we do this you'll see how the concepts we've just covered hang together.

IDENTITY Columns

This has been a big chapter, and we've looked at some really important stuff. We'll continue with our journey into the wacky world of tables in Chapter 5, but we have one more thing to do before our Contacts table is ready for that: we need to add what SQL Server calls an `IDENTITY` column to it.

An `IDENTITY` column auto-populates itself with a numeric value. You configure the `IDENTITY` column's properties to determine what the starting number is and how that number is incremented.

You can create one `IDENTITY` column per table. `IDENTITY` columns are almost always created as the primary key. Using an `IDENTITY` column allows the database to manage primary key values, and removes the responsibility from the developer.

We are going to turn the `ContactId` column into an `IDENTITY` column. We'll start assigning values from number 1, and then we'll increment the number by 1 every time. So our first contact should be numbered 1, the second 2, and so on. The `IDENTITY` column call looks like this:

```
IDENTITY(seed, increment)
```

The `seed` is the starting number, and the `increment` dictates what the next number will be. You could have a seed of 2 and an increment of 5, which would generate values of 2, 7, 12, 17 and so on.

Open up our `Contacts` table script (this should be at `c:\temp\sqlbasics\apply\02 - Create Contacts Table.sql`). Find the `ContactId` line and change it to:

```
ContactId INT IDENTITY(1,1),
```

We've added `IDENTITY(1,1)`. These are actually the default values, so we could have just written `IDENTITY()`. But it is always better to be explicit, as it makes your code easier to read.

Your table script should now look like this:

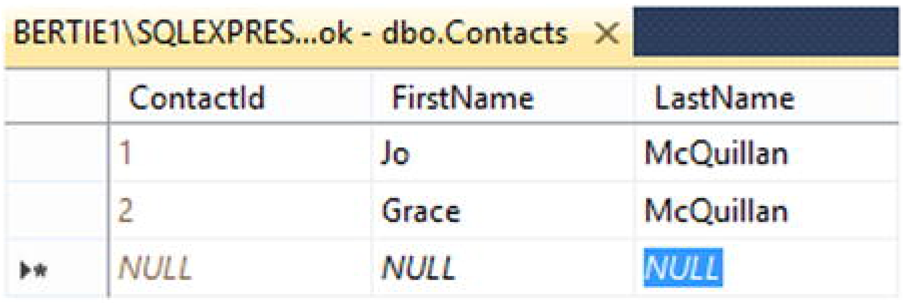
```
USE AddressBook;

IF EXISTS (SELECT 1 FROM sys.tables WHERE [Name] = 'Contacts')
BEGIN
DROP TABLE dbo.Contacts;
END;

CREATE TABLE dbo.Contacts
(
ContactId INT IDENTITY(1,1),
FirstName VARCHAR(40),
LastName VARCHAR(40),
DateOfBirth DATE,
PhoneNumbers VARCHAR(200),
AllowContactByPhone BIT,
FirstAddress VARCHAR(200),
SecondAddress VARCHAR(200),
RoleId INT,
RoleTitle VARCHAR(200),
Notes1 VARCHAR(200),
Notes2 VARCHAR(200),
DrivingLicenseNumber VARCHAR(40),
PassportNumber VARCHAR(40),
ContactVerified BIT,
CreatedDate DATETIME,
CONSTRAINT PK_Contacts PRIMARY KEY CLUSTERED (ContactId)
);
```

```
GO
```

You can easily identify the lines you have changed, as they will have a yellow mark next to them (lines with a green mark are already saved). Press F5 to run this, then open the table for editing (right-click the table in Object Explorer and select **Edit Top 200 Rows**). The eagle-eyed among you may notice the `NULL` value under `ContactId` is a slightly different color than the other `NULL`s. This is because this column's value is now automatically computed for us. Try typing into it—nothing will happen. Now enter a couple of contacts, just the first name and last name (press Enter once you've entered these details for a contact to move to the next line). You'll see the `ContactId` column's value automatically appear, like in Figure 4-11.



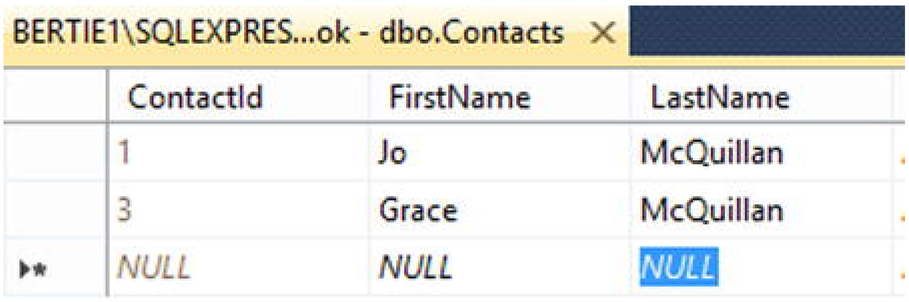
BERTIE1\SQLEXPRES...ok - dbo.Contacts

	ContactId	FirstName	LastName
	1	Jo	McQuillan
	2	Grace	McQuillan
▸*	NULL	NULL	NULL

Figure 4-11. *The `IDENTITY` column in action*

Lovely stuff! Now for one last demonstration. Delete your last contact (so I'd delete `ContactId` 2 in the table shown above). To delete, right-click the gray box just before the number 2 and choose the **Delete** option from the context menu. You'll be asked to confirm the deletion; click **Yes** to delete the row.

Now, add the row back. What `ContactId` number do you think will be allocated? Figure 4-12 has the answer:



BERTIE1\SQLEXPRES...ok - dbo.Contacts

	ContactId	FirstName	LastName
	1	Jo	McQuillan
	3	Grace	McQuillan
▸*	NULL	NULL	NULL

Figure 4-12. *A missing `IDENTITY` value*

If your answer was 2, I'm afraid you can't pass Go and collect £200 (or $200 if you are in the United States). The IDENTITY column does not reuse values, so the next unused value, 3, is assigned instead.

I highlight this to demonstrate that you should never assume an `IDENTITY` column's values are sequential. Events like deleting records or failed inserts will cause gaps in your `IDENTITY` column's values. This isn't a problem, as usually all we are concerned about is the uniqueness of the value.

Now we have everything in place to move on with our design!

Summary

Wow, that was an involved chapter. You've just been through a roller coaster of SQL table details. We began by finding out that a table is the bread and butter of a database, and consists of columns and rows of data. We then met the `CREATE TABLE` statement and saw how SSMS can be used to create tables.

We had a little look at the data types we can store in tables, such as `VARCHAR` and `INT`. We also saw what primary and foreign keys do, why we need them, and how primary keys can be clustered or nonclustered.

Our farewell to this chapter involved finding out what `IDENTITY` columns do. We ended up with an enhance version of our `Contacts` table.

Next, we're going to build on this platform as we continue to refine and improve our **AddressBook** database. Heigh-ho, heigh-ho. . . .

PREV
Chapter 3 : Database Basics

NEXT
Chapter 5 : Putting Good Tables …

R
S