CHAPTER 5

Putting Good Tables Together

Now that you have an idea about how tables work, we need to make sure they can store our data in a proper manner. To do this, we use a process called *Database Normalization*. Once this has been applied to our tables, we should have an efficient database structure that minimizes duplicate data. Or to put it simply, everything should just work!

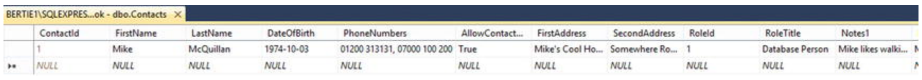
Database Normalization

A database is considered normalized when it meets a set of rules, known as *the three rules of normalization*. All this means is the database is structured in a way that follows recognized good practices and supports efficient data querying. It reduces the amount of duplication in the database and tries to ensure the database is structured correctly.

We'll meet each rule one by one, and apply each to our `Contacts` table. Normalization should be applied on a table-by-table basis, although you'll often find yourself starting from one table and then normalizing the others from there.

First Normal Form

A table is said to meet First Normal Form (1NF) when it doesn't contain any columns that can be used to store repeating groups. What this means is you should not have any columns that store more than one value, or multiple columns storing similar values. Let's look at our `Contacts` table in Figure 5-1, populated with some data.


BERTIE1\SQLEXPRES...ok - dbo.Contacts

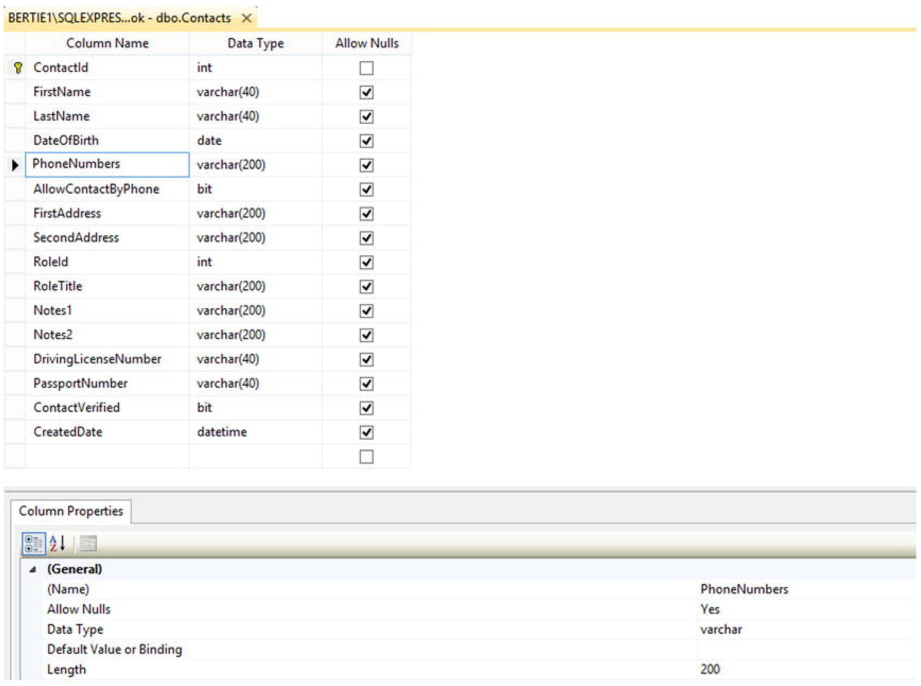
	ContactId	FirstName	LastName	DateOfBirth	PhoneNumbers	AllowContact...	FirstAddress	SecondAddress	RoleId	RoleTitle	Notes1
	1	Mike	McQuillan	1974-10-03	01200 313131, 07000 100 200	True	Mike's Cool Ho...	Somewhere Ro...	1	Database Person	Mike likes walki...
▸*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Figure 5-1. *A Contact record in 1NF*

To add this data, right-click the `Contacts` table in Object Explorer and click **Edit Top 200 Rows**. A grid will appear and you can type in the values in Figure 5-1. You can enter anything you want; just make sure you specify two phone numbers in the `PhoneNumbers` column, separated by a comma.

There are two repeating groups in this table. Can you spot them? Take a look at Figure 5-2 for a full list of columns.

- The `PhoneNumbers` column is being used to store multiple values.
- The `Notes1` and `Notes2` columns limit us to two notes per contact. This is a poor structure and limits our design.



BERTIE1\SQLEXPRES...ok - dbo.Contacts

Column Name	Data Type	Allow Nulls
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	☑
		☐

Column Properties

(General)	
(Name)	PhoneNumbers
Allow Nulls	Yes
Data Type	varchar
Default Value or Binding	
Length	200

***Figure 5-2**. The SSMS table editor*

The `PhoneNumbers` column is being used to store more than one phone number, so it is a repeating group. You can assume that anything being used to store more than one value is a repeating group. The pluralization of the name is also something of a giveaway. To fix this, we should rename the `PhoneNumbers` column to `PhoneNumber`, and then separate the row into two rows.

There are two ways to rename the column: we can use SSMS, or something called a *system stored procedure*. We'll look at both options, as we'll be doing lots of work on stored procedures later in the book.

Let's use SSMS first. Close the table grid if you still have it open—you cannot edit the table if it is open (also, if other users have your table open you'll be prevented from editing it). Right-click the `Contacts` table and select the **Design** option. The table editor in Figure 5-2 will appear.

You can edit any column in the table here. Clicking a column name will allow you to rename the column, and clicking in a Data Type cell will let you change the data type for a column. The Allow Nulls column determines whether null values can be stored in the column or not. A null value simply means no value is required for this column; it is optional. We'll see much more of `NULL` in the rest of the book.

You can also change settings by right-clicking columns (see Figure 5-3). If you right-click the little gray square next to `ContactId`, for example, a dialog appears. From this dialog you can set the selected column as the primary key, insert or delete columns, and create indexes. Note that `ContactId` has a key next to it, denoting it is already set as the primary key (we did this in Chapter 4). Because of this, the menu option asks us if we want to *remove* the primary key, not set it.

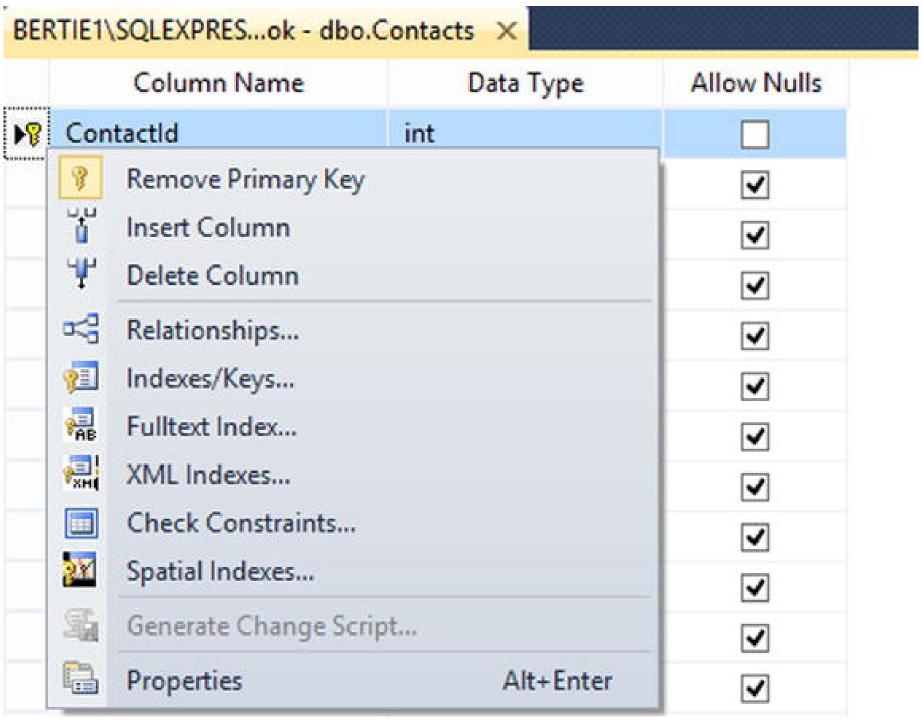
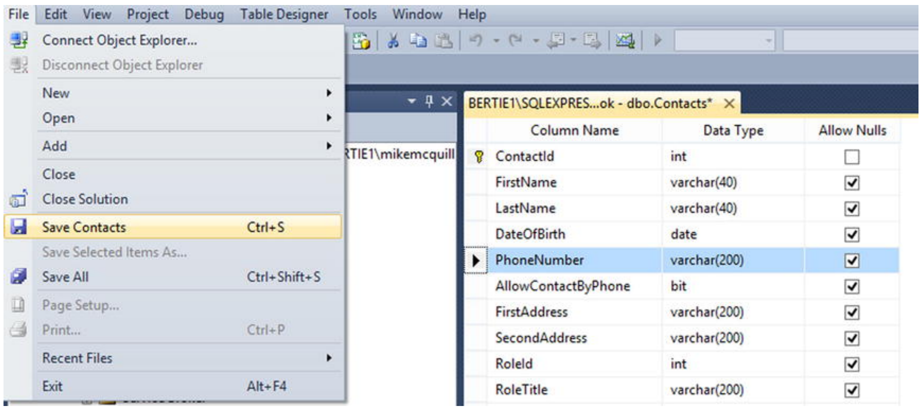


Figure 5-3. Column options in the SSMS table editor

For now, we just want to rename the `PhoneNumbers` column. Click `PhoneNumbers` and remove the `s` at the end, so it becomes `PhoneNumber`. After doing this, go to the **File** menu and click **Save Contacts**, as shown in Figure 5-4.



Column Name	Data Type	Allow Nulls
ContactId	int	☐
FirstName	varchar(40)	✓
LastName	varchar(40)	✓
DateOfBirth	date	✓
PhoneNumber	varchar(200)	✓
AllowContactByPhone	bit	✓
FirstAddress	varchar(200)	✓
SecondAddress	varchar(200)	✓
RoleId	int	✓
RoleTitle	varchar(200)	✓

Figure 5-4. Saving table changes in SSMS

Note the asterisk in the yellow bar next to `dbo.Contacts`. This means there are unsaved changes. Once you click **Save Contacts** the asterisk should disappear. Click the **X** to the right of the yellow bar to close the table editor. Now right-click the `Contacts` table and choose **Edit Top 200 Rows**. The `PhoneNumbers` column is no more—it is now `PhoneNumber`!

Remember what we've been talking about though; you can't give a DBA instructions on what they have to do in SSMS—they expect a script! It is not possible using standard SQL statements to rename a column. You would have to create the new column, copy the data into it, and then drop the existing column. Fortunately, we don't need to do this. SQL Server provides a rename feature, wrapped up in a system stored procedure called `sp_rename`.

WHAT ARE SYSTEM STORED PROCEDURES?

We'll see how you can create your own stored procedures in Chapter 18, but SQL Server comes with a lot of built-in stored procedures, known as *system stored procedures*. These usually begin with `sp_`, as we've just seen with `sp_rename`. There are system stored procedures that help you manage sending e-mail from the database, and procedures that tell you about objects in your database (`sp_help`).

Visit `https://msdn.microsoft.com/en-us/library/ms187961.aspx` for more information about system stored procedures.

The `sp_rename` procedure allows us to rename certain objects, like tables, stored procedures, and columns. To put the `PhoneNumbers` column back we'd write:

```
USE AddressBook;

EXEC sp_rename
@objname = 'dbo.Contacts.PhoneNumber',
@newname = 'PhoneNumbers',
@objtype = 'COLUMN';

GO
```

Note that when we are specifying the existing column name (the `@objname` parameter), we put the table name in front of it, but not for the `@newname` parameter. We need to specify the table name so the stored procedure knows which table to look in. After running this you'll see this message:

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

The message is quite right—you should be very careful about renaming things in existing databases. If you were to rename a database object that is being used in a production system, you'd need to make sure everything that referenced the old name was updated.

We've restored the name to the non-normalized version. You can either design or edit the table to see this. Now we'll correct the name again, by slightly changing the values we are passing to the stored procedure:

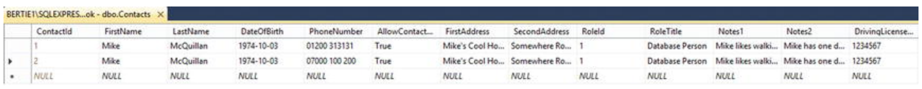
```
USE AddressBook;

EXEC sp_rename
@objname = 'dbo.Contacts.PhoneNumbers',
@newname = 'PhoneNumber',
@objtype = 'COLUMN';

GO
```

Don't forget to open `c:\temp\sqlbasics\apply\02 - Create Contacts Table.sql` and rename the `PhoneNumbers` column to `PhoneNumber`—we need to keep our script up to date. Fixing the column name in the database will not automatically fix our script for us, and we'll be using that script again very soon.

The `PhoneNumber` column now meets our 1NF requirements, but the data doesn't. We need to split our current row into two rows, one per phone number. Your data should look like Figure 5-5.



	ContactId	FirstName	LastName	DateOfBirth	PhoneNumber	AllowContact...	FirstAddress	SecondAddress	RoleId	RoleTitle	Notes1	Notes2	DrivingLicense...
	1	Mike	McQuillan	1974-10-03	01200 313131	True	Mike's Cool Ho...	Somewhere Ro...	1	Database Person	Mike likes walki...	Mike has one d...	1234567
▸	2	Mike	McQuillan	1974-10-03	07000 100 200	True	Mike's Cool Ho...	Somewhere Ro...	1	Database Person	Mike likes walki...	Mike has one d...	1234567
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

__Figure 5-5__. Contacts data that meets 1NF

There's a lot of duplication there, especially around the `Notes1` and `Notes2` columns. Remember that we had two repeating groups: `PhoneNumbers` and `Notes`. We'll see what we can do about the repeating `Notes` group.

Multiple-Column Repeating Groups

The `PhoneNumbers` column was a single column housing multiple values, so it was easy to bring it to 1NF; all we needed to do was split the rows. You might still think the fix we've made for `PhoneNumbers` is not quite right, but we'll look at that further on down the road.

The fix for multiple-column repeating groups is not as straightforward. We have two columns:

- `Notes1`
- `Notes2`

If you see a table using a similar approach, I can guarantee in no time at all you'll be asked to add a `Notes3` column. If you need more than one column to represent the same thing, then you should be putting those columns into their own tables. That's how we'll solve this problem. We'll then link the new table back to the `Contacts` table via the `ContactId` primary key. This mechanism will allow us to store as many notes as we want for each contact, because we'll create a one-to-many relationship (one contact can have one or more notes). As we put this into practice we'll see how to:

- create a table using T-SQL,
- create a foreign key using the database diagramming tool, and
- use SSMS to auto-generate the T-SQL for our foreign key.

What do we need in our table? Well, as this is going to be a child table of `Contacts`, we can immediately see we need a `ContactId` column. This will not be an `IDENTITY` column; otherwise we couldn't link it back to the appropriate record in `Contacts`. We also need a `NoteId` column of some sort, allowing us to uniquely identity a note. This will be the primary key and will be configured as an `IDENTITY` column. Finally, we'll need a `Notes` column to store the actual note details. As both existing `Notes` columns are `VARCHAR(200)`, that's what we'll use as the data type.

Open a New Query Window (Ctrl+N) and type the following script. We will call this table `ContactNotes`.

```
USE AddressBook;

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

CREATE TABLE dbo.ContactNotes
(
NoteId INT IDENTITY(1,1),
ContactId INT,
Notes VARCHAR(200),
CONSTRAINT PK_ContactNotes PRIMARY KEY CLUSTERED (NoteId)
);

GO
```

This script doesn't present anything we haven't seen before. We start with what should be our now familiar `IF EXISTS...DROP TABLE` check. We follow this with a `CREATE TABLE` statement. Nothing exciting. Now we have two tables: `Contacts` and `ContactNotes`.

Save the `ContactNotes` table script as `c:\temp\sqlbasics\apply\03 - Create ContactNotes Table.sql`.

We need to join the tables together, so the database can control which notes belong to which contact. We'll do this via the database diagramming tool. This is a useful little item built into SSMS, and it's great for laying out your tables so you can easily see how they relate to each other.

Expand the **AddressBook** database in the Object Explorer. The first item under **AddressBook** should be **Database Diagrams**. Right-click this and choose **New Database Diagram** (shown in Figure 5-6).

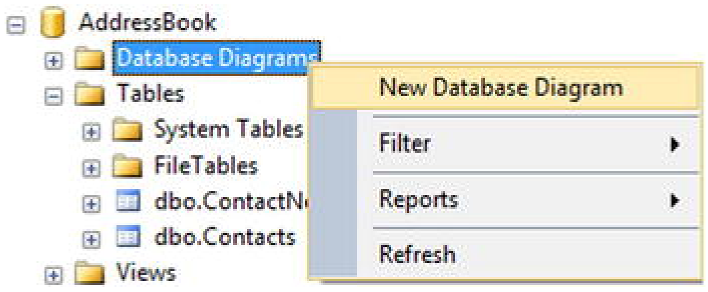


Figure 5-6. *Creating a new database diagram*

When you first click this you may be asked to add database diagramming support to the database. Click **Yes** to do this if you are asked. You should see a screen looking something like Figure 5-7.

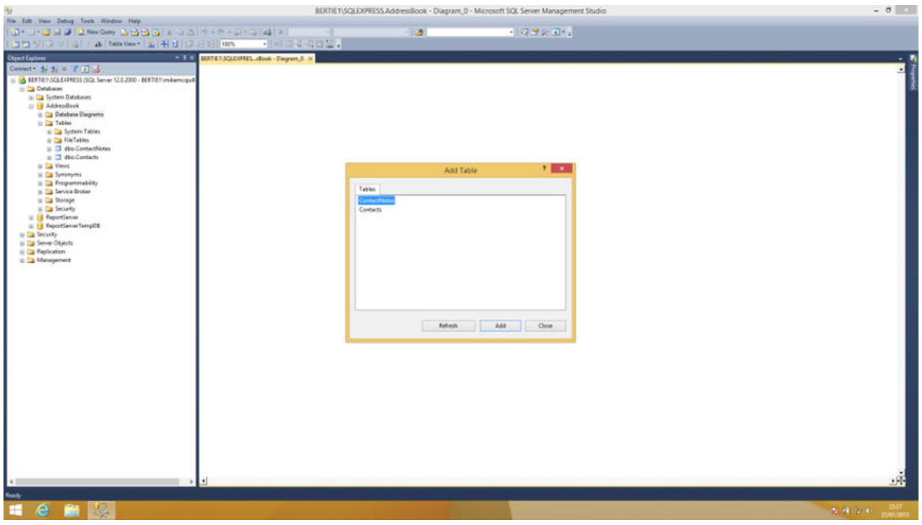

BERTIE1\SQLEXPRESS.AddressBook - Diagram_0 - Microsoft SQL Server Management Studio
Object Explorer
Add Table
Tables
ContactNotes
Contacts
Refresh
Add
Close

Figure 5-7. *Adding tables to a database diagram*

The Add Table dialog in front lists all the tables available in the database. The big blank area behind it is the diagram pane.

REFRESHING THE DIAGRAM TABLES DIALOG

There is a long-standing issue with the Add Table dialog: it doesn't automatically refresh. If we added another table now and then came back to the diagram to add it, the table would not be visible in the Add Table dialog. You need to manually refresh the list by clicking the Refresh button.

To add a table to a diagram, highlight the table name and click the **Add** button. We only have two tables, so just click **Add** twice, then click **Close**. The tables will be displayed on the diagram, which should look similar to Figure 5-8.

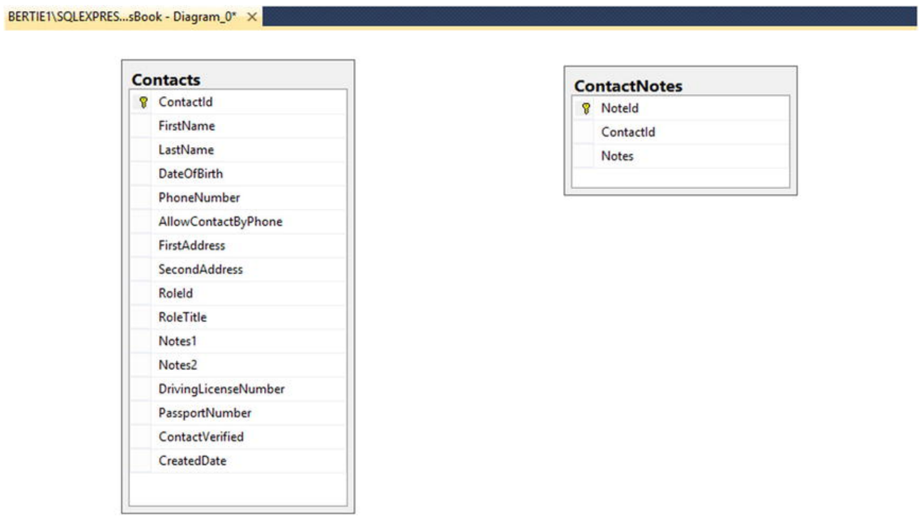
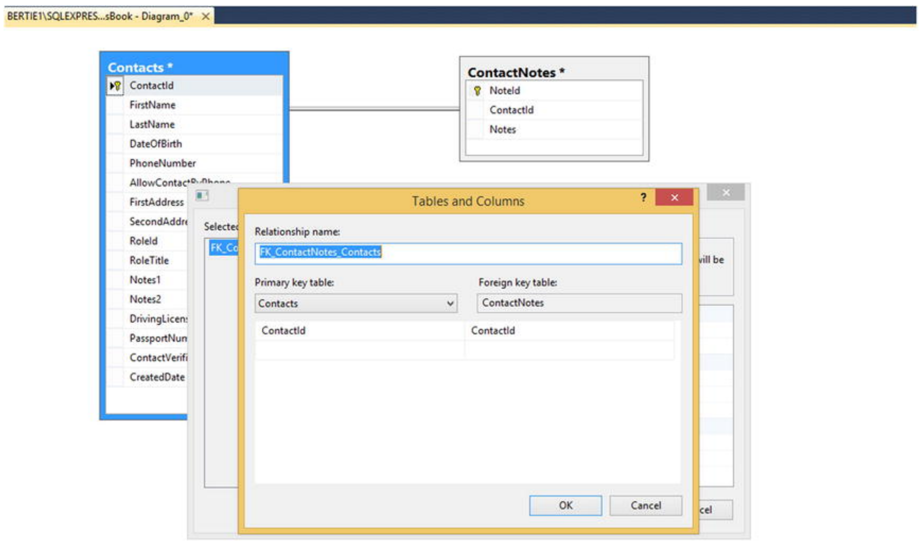


Figure 5-8. *Database diagram with unconnected tables*

The diagramming tool doesn't always lay out tables in the best position, so don't be surprised if the diagram layout is all over the place! You can drag a table anywhere within the diagram pane by dragging it using its ti

At the moment, our two tables are completely unrelated. We are going to change this. We want to make `Contacts` a parent of `ContactNotes`. That is, we are saying a `ContactNotes` record cannot exist unless it is linked to a `Contact` record. The column both tables share is `ContactId`. Place your mouse cursor over the grey box next to the `ContactId` column in the `Contacts` table. Hold down the left mouse button and drag the cursor over to the `ContactId` column in the `ContactNotes` table. A dotted line appears, and when you move the cursor over the `ContactId` column in `ContactNotes` it gains a small + sign, denoting a relationship can be created.

Once your cursor is positioned over the `ContactId` column in `ContactNotes`, a dotted line will appear. Release the mouse button and wonderful things will happen. Okay, they won't, but you will see a Tables and Columns dialog as per Figure 5-9.



Tables and Columns

Relationship name:
FK_ContactNotes_Contacts

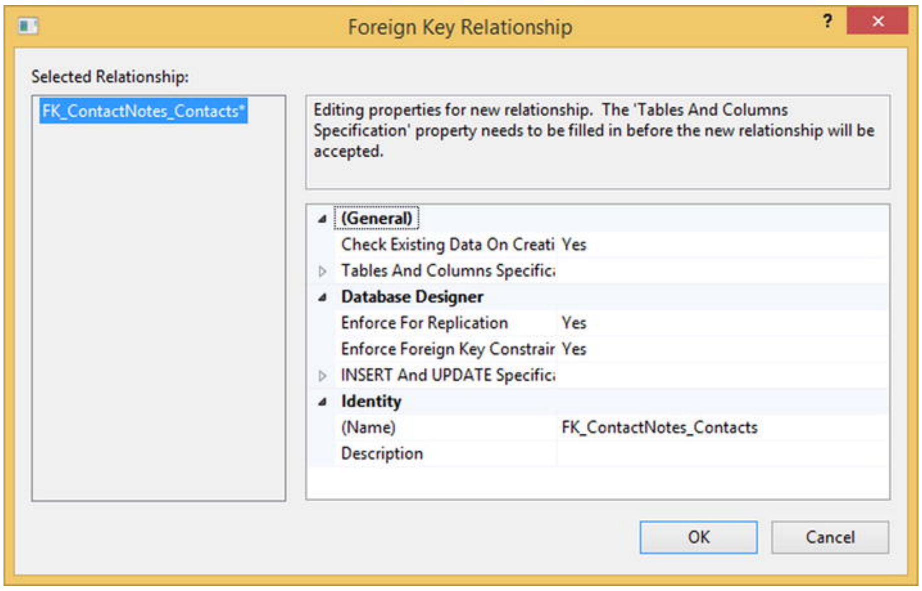
Primary key table:	Foreign key table:
Contacts	ContactNotes
ContactId	ContactId

Figure 5-9. Creating a foreign key in the Database Diagramming tool

We can see that SQL Server has deduced a relationship name for us: `FK_ContactNotes_Contacts`. This name is fine and makes sense. FK tells us it is a foreign key, and the rest of it tells us the names of the two tables involved in the relationship, separated by underscores. This is a good convention to use for naming your foreign keys, and it's one we'll return to as we create more relationships.

The dialog also shows us what the **Primary key table** (the parent) is in the relationship, and also the **Foreign key table** (a.k.a. the child). The columns to be linked are shown underneath the tables.

Click the **OK** button. The dialog will disappear and the properties dialog for the Foreign Key Relationship, which has been hiding behind the Tables and Columns dialog, will reveal itself (Figure 5-10).


Foreign Key Relationship

Selected Relationship:

FK_ContactNotes_Contacts*

Editing properties for new relationship. The 'Tables And Columns Specification' property needs to be filled in before the new relationship will be accepted.

(General)	
Check Existing Data On Creati	Yes
Tables And Columns Specific:	
Database Designer	
Enforce For Replication	Yes
Enforce Foreign Key Constrair	Yes
INSERT And UPDATE Specific:	
Identity	
(Name)	FK_ContactNotes_Contacts
Description	

Figure 5-10. *Foreign Key Relationship properties*

There are some interesting things to look at here, as you can tailor quite a few properties for the relationship.

- **Check Existing Data On Creation Or Re-Enabling**: If the tables already have data in them, enabling this option will cause the data to be validated when the relationship is created. If data exists that will not support the relationship (e.g., some `ContactId`s in `ContactNotes` that do not exist in `Contacts`), an error will appear and the relationship will not be created until the offending data has been corrected or removed.
- **INSERT And UPDATE Specification**: If you expand this you'll see:
  - **Delete Rule**: No Action
  - **Update Rule**: No Action

    This means that if a record in the `Contacts` table is deleted, any `ContactNotes` for that contact will not be deleted. Similarly, if the `Contacts` record is updated the corresponding `ContactNotes` records will not be updated. We can change this setting. The options are:

  - **No Action**: Do nothing. If you try to delete a parent record that has associated children, an error message will be displayed explaining that you cannot delete the parent until the children have been deleted.
  - **Cascade**: If the parent record is deleted, the children are deleted. If the parent record is updated, the children are updated, too.
  - **Set Null**: If the parent record is deleted, set the foreign key column (`ContactId` in `ContactNotes` in this case) to `NULL`. The same applies to updates if the primary key of the parent table can be manually updated.
  - **Set Default**: Same as Set Null, except the column's default value is set on the child records (I'll introduce default column values in the very next chapter). If no default value is set, the foreign key column will be set to `NULL`.

We won't change any settings. Click **OK** to view your new diagram. You should see the line shown in Figure 5-11 has been added, linking `Contacts` to `ContactNotes`.

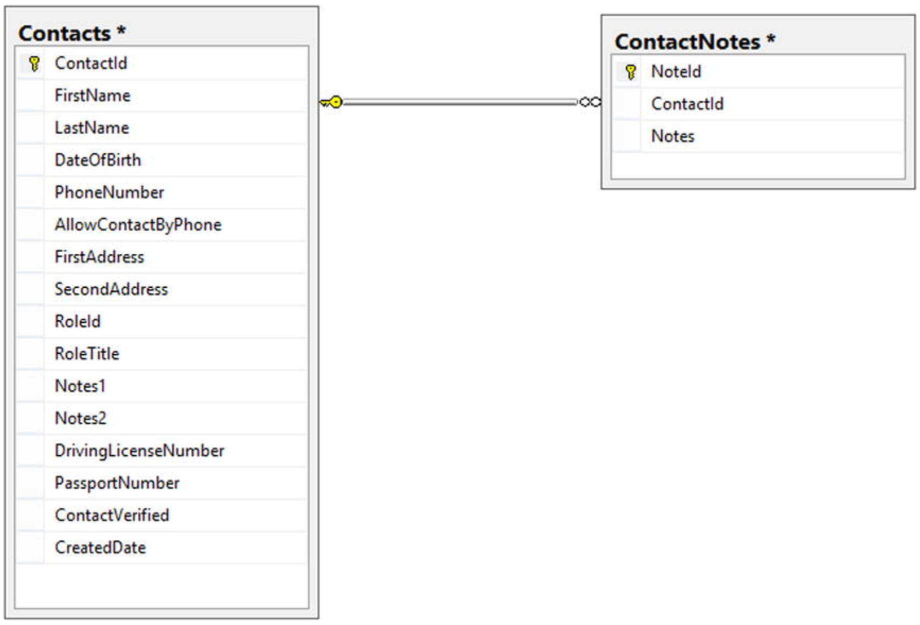


Figure 5-11. *Database diagram with a one-to-many relationship*

Note the asterisks next to the table names. These indicate the relationship hasn't been saved yet. You should recognize this diagram; it looks similar to the diagrams we saw in the last chapter when we began to investigate foreign keys.

The left-hand side of the line—the `Contacts` side—begins with a key. This denotes the primary key or parent side of the relationship (the "one" of "one to many"). The right-hand side has the figure eight present, denoting the "many" side—the foreign key or child side.

Go to the **File** menu and click **Save Diagram_0**. You will be asked to give the diagram a name—call it **Contacts**. Click **OK** and you'll see another dialog asking you to confirm the save operation. If you don't want to see this again, uncheck the **Warn about Tables Affected** box in the bottom left-hand corner. Click **Yes** to save the changes. The stars next to the names will disappear and your relationship will be saved.

There's still an issue to resolve: we haven't scripted the creation of our foreign key. Your DBAs won't be pleased with you if you ask them to use the diagramming tool to create a foreign key!

Fortunately, SSMS can come to our rescue here. In the Object Explorer, expand **Databases**, **AddressBook**, **Tables**, **ContactNotes**, and **Keys**. You should see two entries under here: one for the primary key, and one for the foreign key we just created. Right-click the foreign key `FK_ContactNotes_Contacts`, go to **Script Key as**, then **CREATE To**, and finally **New Query Editor Window**. The full menu is shown in Figure 5-12.

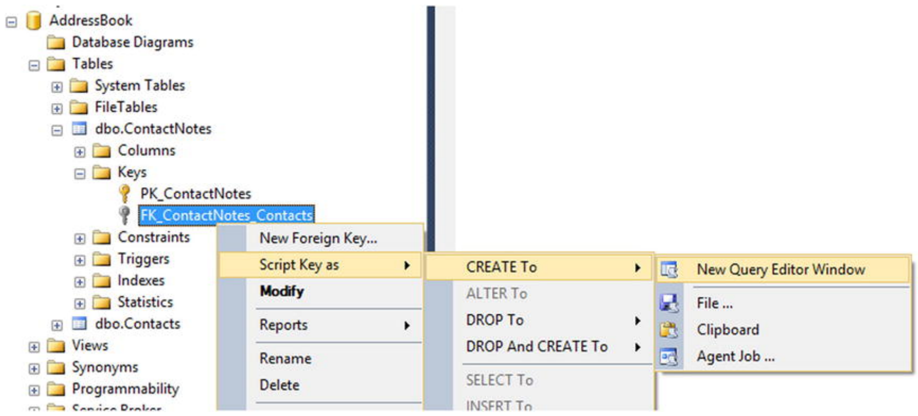


Figure 5-12. *Scripting a foreign key from SSMS*

The query window will appear, populated with a couple of `ALTER TABLE` statements:

```
USE [AddressBook];
GO

ALTER TABLE [dbo].[ContactNotes]
WITH CHECK ADD CONSTRAINT [FK_ContactNotes_Contacts] FOREIGN KEY([ContactId])
REFERENCES [dbo].[Contacts] ([ContactId]);
GO

ALTER TABLE [dbo].[ContactNotes] CHECK CONSTRAINT [FK_ContactNotes_Contacts];
GO
```

We are only interested in the first `ALTER TABLE` statement; the second can be ignored. (We'll look at check constraints in Chapter 7.) Tidy up the preceding script so it looks like this:

```
ALTER TABLE dbo.ContactNotes
WITH CHECK
ADD CONSTRAINT FK_ContactNotes_Contacts
FOREIGN KEY (ContactId)
REFERENCES dbo.Contacts (ContactId)
```

We'll have a quick walk through this `ALTER TABLE` statement:

- `ALTER TABLE dbo.ContactNotes` tells us that we are altering the ContactNotes table.
- `WITH CHECK` means the data in the `ContactNotes` table will be checked to ensure it meets the relationship requirements (i.e., the `ContactId` column only contains values that are present in the `Contacts` table).
- `ADD CONSTRAINT FK_ContactNotes_Contacts` tells SQL Server we want to create a constraint with the name given.
- `FOREIGN KEY (ContactId)` tells SQL Server the type of constraint we want to create—in this instance, a foreign key. By putting `(ContactId)` after this we are stating our foreign key will use the `ContactId` column in the ContactNotes table.
- `REFERENCES Contacts (ContactId)` completes the statement by adding the parent side of the foreign key relationship. We are linking to the `Contacts` table via its `ContactId` column.

We need to cut and paste this statement into our `ContactNotes` script. We cannot add it to the `Contacts` script, as when that runs the `ContactNotes` table would not exist. (We're going to look at the order the scripts run in once we've finished our normalization process.)

Cut the `ALTER TABLE` statement and paste it into the `c:\temp\sqlbasics\apply\03 - Create ContactNotes Table.sql` script. Paste the `ALTER TABLE` statement between the `CREATE TABLE` and `GO` statements. The three statements should look like this:

```
CREATE TABLE dbo.ContactNotes
(
NoteId INT IDENTITY(1,1),
ContactId INT,
Notes VARCHAR(200),
CONSTRAINT PK_ContactNotes PRIMARY KEY CLUSTERED (NoteId)
);

ALTER TABLE dbo.ContactNotes
WITH CHECK
ADD CONSTRAINT FK_ContactNotes_Contacts
FOREIGN KEY(ContactId)
REFERENCES dbo.Contacts (ContactId);

GO
```

We're nearly at 1NF! Just one thing left to do: remove the `Notes1` and `Notes2` columns from the `Contacts` table. Now that we have the `ContactNotes` table we don't need them any longer. We can write a quick script to do this. Open a New Query Window and type the following:

```
USE AddressBook;
```

```
ALTER TABLE dbo.Contacts
DROP COLUMN Notes1, Notes2;

GO
```

DROPPING COLUMNS IN T-SQL

If you need to drop a column using T-SQL, you use the `ALTER TABLE` statement, as this code shows. You can specify as many columns as you want when dropping columns in this way, but obviously all columns must exist in the table for the command to execute successfully.

Execute the script and then close it (you don't need to save it). If you close your diagram and reopen it, you'll see the `Notes1` and `Notes2` columns no longer exist.

We need to remove these columns from the main table script. Open up `c:\temp\sqlbasics\apply\02 - Create Contacts Table.sql` and remove the following lines:

```
Notes1 VARCHAR(200),
Notes2 VARCHAR(200),
```

This ensures we won't accidentally add the columns back in when we run the scripts in one go. Your `CREATE TABLE` statement should now look like this:

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

Woohoo! We are at 1NF—we've eliminated all repeating groups. Let's step up a level to 2NF.

Second Normal Form

Second Normal Form, or 2NF, is met if the table is in 1NF and all non-key attributes are dependent upon the primary key. That is, all columns in a row depend upon the primary key for their existence. Our primary key is `ContactId`, so all other columns should be able to be derived from that. `FirstName`, `LastName`, `DateOfBirth`, `PhoneNumber`, and `AllowContactByPhone` all seem fine—they certainly belong to a contact. Same for `DrivingLicenseNumber`, `PassportNumber`, and `ContactVerified` (and `CreatedDate` is just a bit of metadata—we'll discuss that soon). `FirstAddress` and `SecondAddress` are dependent upon a contact, although they are a bit ambiguous. Do these represent the first and second lines of an address, as I used them earlier? Or are they supposed to store an address each, such as a home address and a work address? If it's the second option, then this is a repeating group, which we should have resolved during the 1NF phase. For now, we'll assume these are OK.

Next, we have `RoleId`. I'm not convinced that is reliant on the `ContactId`. A contact may have a role, but does that mean we care about the `RoleId`? The next column confirms our thoughts here: `RoleTitle`. This is what we are interested in for our contacts, but surely `RoleTitle` is dependent upon `RoleId`?

`RoleId` and `RoleTitle` are preventing us from meeting 2NF. We want to be able to link `Contacts` to `Roles`, but we should not be storing these pieces of information in the same table. What we need to do to fix this problem is the following:

- Move `RoleId` and `RoleTitle` out into their own table, `Roles`.
- Create a new table, `ContactRoles`, to store a `ContactId` and `RoleId`.
- Create a one-to-many relationship from `Contacts` to `ContactRoles`.
- Create a second one to many relationship from `Roles` to `ContactRoles`.

You may be able to tell from this description that we are going to create a many-to-many relationship between `Contacts` and `Roles`; that is, one contact may be assigned to multiple roles, and one role may be assigned to multiple `Contacts` (e.g., your firm might employ 20 SQL developers).

Open a New Query Window and type in the script for the `Roles` table.

```
USE AddressBook;

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

CREATE TABLE dbo.Roles
(
RoleId INT IDENTITY(1,1),
RoleTitle VARCHAR(200),
CONSTRAINT PK_Roles PRIMARY KEY CLUSTERED (RoleId)
);

GO
```

Nothing we haven't seen here before. We add the usual `IF EXISTS` check, then create the table. We create a clustered primary key, so the roles will be sorted on the disk by `RoleId`. This makes sense as we are creating a many-to-many relationship using this column.

Run this script and save it as `c:\temp\sqlbasics\apply\04 - Create Roles Table.sql`. Open another New Query Window and add the script for the `ContactRoles` table.

```
USE AddressBook;

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

CREATE TABLE dbo.ContactRoles
(
ContactId INT,
RoleId INT,
CONSTRAINT PK_ContactRoles PRIMARY KEY CLUSTERED (ContactId, RoleId)
);

GO
```

This is standard stuff until we come to the primary key declaration. The primary key is made up of two columns —a compound key.

What I am saying here is each unique combination of a contact and a role can only exist once in the table. Grace McQuillan can be a developer and she can be a DBA, too, but she can't be a developer and a developer.

Another first for us: this is the first table we've created that contains primary key columns and nothing else. Quite often a many-to-many table will only house primary key columns, as its whole purpose is to provide the link between two other tables.

We're not quite done with this script yet—we need to add the foreign key references to `Contacts` and `Roles`. You're going to type these manually this time. Let's do the `Contacts` foreign key first. After the `CREATE TABLE` statement (but above the `GO` statement), type the `ALTER TABLE` statement.

```
ALTER TABLE dbo.ContactRoles
ADD CONSTRAINT FK_ContactRoles_Contacts
```

```
FOREIGN KEY (ContactId)
REFERENCES dbo.Contacts (ContactId)
ON UPDATE NO ACTION
ON DELETE CASCADE;
```

Hmm, this looks slightly different from what we saw earlier. The first four lines are the same, but the last two lines are new. Remember the **INSERT And UPDATE Specification** from the diagramming tool earlier? These lines map to what that was doing.

- **`ON UPDATE NO ACTION`** states that if we update the `Contacts` table, we won't make any changes to the `ContactRoles` table. It is not possible to manually update a `ContactId` value in the `Contacts` table (it's an `IDENTITY` column). As a result, we cannot update it and subsequently cannot update the `ContactRoles` table via the foreign key, so there is no point in specifying a different option for updates.
- **`ON DELETE CASCADE`** tells SQL Server to delete any `ContactRoles` records should we delete the parent contact record.

We'll add a very similar statement to create the relationship between `Roles` and `ContactRoles`. Paste this after the previous `ALTER TABLE` statement (just above the `GO`):

```
ALTER TABLE dbo.ContactRoles
ADD CONSTRAINT FK_ContactRoles_Roles
FOREIGN KEY (RoleId)
REFERENCES dbo.Roles (RoleId)
ON UPDATE NO ACTION
ON DELETE CASCADE;
```

This is exactly the same as above, except we are linking to `Roles` instead of `Contacts`.

Save the script as `c:\temp\sqlbasics\apply\05 - Create ContactRoles Table.sql`, and run it. You should now have four tables in the database:

- `Contacts`
- `Roles`
- `ContactNotes`
- `ContactRoles`

We need to remove the `RoleId` and `RoleTitle` columns from the `Contacts` table. Open the script `c:\temp\sqlbasics\apply\02 - Create Contacts Table.sql`. Remove these two lines:

```
RoleId INT,
RoleTitle VARCHAR(200),
```

Save this script, and open a New Query Window. We need to manually remove the columns from the table. Execute the following `DROP COLUMN` script in a New Query Window—there's no need to save it:

```
USE AddressBook;

ALTER TABLE dbo.Contacts
DROP COLUMN RoleId, RoleTitle;
```

We'll add our new tables to our diagram, so we can see how everything is linking together. Expand **Database Diagrams** and open the `dbo.Contacts` diagram we created earlier. The `Contacts` and `ContactNotes` tables should already be present. Right-click a white area and click **Add Table**. Click **Refresh** on the dialog that appears, then keep clicking **Add** to add the `ContactRoles` and `Roles` tables. Then click the **Close** button.

The diagramming tool is not the greatest when it comes to automatically positioning tables. Drag the tables to a position where you can easily see them and the layout of the relationships. Figure 5-13 shows a nicely laid-out diagram:

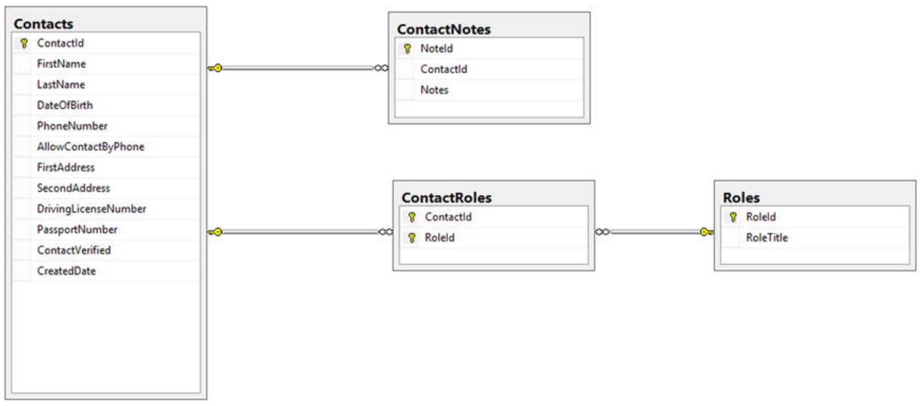


Figure 5-13. *Database diagram with a many-to-many relationship*

You can clearly see how `ContactRoles` enforces the many-to-many relationship. The `Contacts` table has drastically changed from our first attempt; we split the columns we've removed from it into three new tables.

We have now reached base camp, also known as 2NF. All non-key attributes are now dependent upon the primary key. Now it's time to head for the summit—yes, it's 3NF time!

Third Normal Form

There are levels of normalization beyond Third Normal Form (3NF), but these are largely theoretical. We'll be stopping our normalization journey here. A table is considered to meet 3NF if it is already in 2NF and every column in the table is dependent upon the primary key and only the primary key. This sounds a bit similar to 2NF, but there is a difference. 2NF was concerned with eliminating columns that depended upon another column, such as `RoleTitle` being dependent on `RoleId`. What we are looking for now are columns that don't have a clear dependency on a possible primary key column.

All of our tables except `Contacts` are pretty simple, and successfully meet 3NF already. Looking at the `Contacts` table now, it is clear that most columns are dependent upon the `ContactId` primary key. What are not clear are the dependencies of `FirstAddress` and `SecondAddress`.

If we assume `FirstAddress` should hold the first line of the address and `SecondAddress` should hold the second line of the address, we can deduce that `SecondAddress` has a dependency on `FirstAddress`. This does not meet 3NF. There is an additional issue here: the columns have been named badly, so another developer might think the `FirstAddress` column stores one entire address (e.g., a home address), and the `SecondAddress` column should store a second, entirely different address (e.g., a work address).

We need to resolve the address problems. It is entirely possible that a contact may have more than one address, so we need to move address details into their own table. We can then join this table up to `Contacts` using a one-to-many relationship.

Open a New Query Window and type this script to create the `ContactAddresses` table:

```
USE AddressBook;

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

CREATE TABLE dbo.ContactAddresses
(
AddressId INT IDENTITY(1,1),
ContactId INT,
HouseNumber VARCHAR(200),
Street VARCHAR(200),
City VARCHAR(200),
Postcode VARCHAR(20),
CONSTRAINT PK_ContactAddresses PRIMARY KEY NONCLUSTERED (AddressId)
);

GO
```

After the standard `IF EXISTS` check we create the table. We've made a few changes to how addresses are stored. The `AddressId` acts as the primary key and `ContactId` will act as a foreign key, once we've added the relationship. Note that we've declared a `NONCLUSTERED` primary key here—more on this later. We've then added columns to store `HouseNumber`, `Street`, `City`, and `Postcode`. This is much more specific than the address detail we've seen so far. We finish up by declaring `AddressId` as the primary key. All that is left to do is add the foreign key, just above `GO`:

```
ALTER TABLE dbo.ContactAddresses
ADD CONSTRAINT FK_ContactAddresses_Contacts
FOREIGN KEY (ContactId)
REFERENCES dbo.Contacts (ContactId)
ON UPDATE NO ACTION
ON DELETE CASCADE;
```

This links `ContactAddresses` to `Contacts`. Again, we do nothing on an update, but we will delete any records in `ContactAddresses` linked to a `Contacts` record that we delete.

Save this script as `c:\temp\sqlbasics\apply\06 - Create ContactAddresses Table.sql`. Then press F5 to run it.

We need to open `c:\temp\sqlbasics\apply\02 - Create Contacts Table.sql` and remove the `FirstAddress` and `SecondAddress` lines:

```
FirstAddress VARCHAR(200),
SecondAddress VARCHAR(200),
```

Save and close this. We then need to open up a New Query Window and alter the table to remove the columns (again, no need to save this).

```
USE AddressBook;

ALTER TABLE dbo.Contacts
DROP COLUMN FirstAddress, SecondAddress;
```

Return to your diagram and add this table should you wish. We are now at 3NF and our database is ready to use!

Further Analysis

Let's take a look at the completed diagram in Figure 5-14, so we can see what we have.

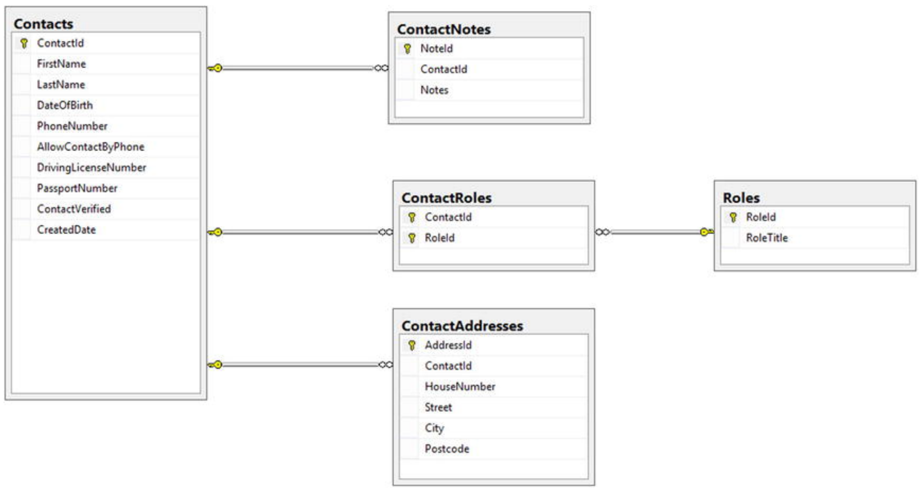


Figure 5-14. *Fully normalized database diagram*

This is looking pretty good. However, I think there are one or two more improvements we can make. These won't affect normalization (all of our tables are now normalized), but they will give our database more flexibility.

Do you remember the first thing we did when we started to normalize our database? We renamed the `PhoneNumbers` column to the singular `PhoneNumber`, and we separated out the row with the comma-separated phone numbers into two rows. What this structure does is prevent a contact from having two phone numbers. If we were to do this in the `Contacts` table now, the same contact would have two different `ContactId` values, so in essence you would have two different `Contacts` with the same name, but different phone numbers.

To enhance this structure we will do the following:

- Remove the `PhoneNumber` column from the `Contacts` table
- Add a new table, `PhoneNumberTypes`, to store the types of phone number.
- Add a second new table, `ContactPhoneNumbers`, to store the phone numbers. We'll add a foreign key to `ContactPhoneNumbers` to store the type of number.

NEVER PUT EARLY DESIGN DECISIONS OFF

You could argue we don't need to make the `PhoneNumber` change. But if you spot something during your design phase that you think will become a limitation later on, think about it and fix it straightaway. This will save you a lot of pain in the long run!

Open up the script `c:\temp\sqlbasics\apply\02 - Create Contacts Table.sql` and remove this line:

```
PhoneNumber VARCHAR(200),
```

This should leave the `CREATE TABLE` statement looking thus:

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

Save this, then open a New Query Window and write an `ALTER TABLE` statement to remove the `PhoneNumber` column from the `Contacts` table.

```
ALTER TABLE dbo.Contacts
DROP COLUMN PhoneNumber;
```

Close this window and open another New Query Window, then add this script to create the `PhoneNumberTypes` table.

```
USE AddressBook;

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

CREATE TABLE dbo.PhoneNumberTypes
(
PhoneNumberTypeId TINYINT IDENTITY(1,1),
PhoneNumberType VARCHAR(40),
CONSTRAINT PK_PhoneNumberTypes PRIMARY KEY CLUSTERED (PhoneNumberTypeId)
);

GO
```

The `CREATE TABLE` statement here is pretty simple; the only new thing here is we've used a `TINYINT` data type instead of an `INT`. `TINYINT` can store up to 255 values, so this would allow us 255 phone number types, which is plenty. Save this script as `c:\temp\sqlbasics\apply\07 - Create PhoneNumberTypes Table.sql` before executing it.

Open another New Query Window and enter this script to create the `ContactPhoneNumbers` table.

```
USE AddressBook;

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

CREATE TABLE dbo.ContactPhoneNumbers
(
PhoneNumberId INT IDENTITY(1,1),
ContactId INT,
PhoneNumberTypeId TINYINT,
PhoneNumber VARCHAR(30),
CONSTRAINT PK_ContactPhoneNumbers PRIMARY KEY CLUSTERED (PhoneNumberId)
);

ALTER TABLE dbo.ContactPhoneNumbers
ADD CONSTRAINT FK_ContactPhoneNumbers_Contacts
FOREIGN KEY (ContactId)
REFERENCES dbo.Contacts (ContactId)
ON UPDATE NO ACTION
ON DELETE CASCADE;

ALTER TABLE dbo.ContactPhoneNumbers
ADD CONSTRAINT FK_PhoneNumberTypes_ContactPhoneNumbers
FOREIGN KEY (PhoneNumberTypeId)
REFERENCES dbo.PhoneNumberTypes (PhoneNumberTypeId)
ON UPDATE NO ACTION
ON DELETE CASCADE;

GO
```

Note that the `PhoneNumberTypeId` column is declared as `TINYINT`, so it matches the primary key in the `PhoneNumberTypes` table. We finish off by creating the two relationships we need.

Save this as `c:\temp\sqlbasics\apply\08 - Create ContactPhoneNumbers Table.sql`. Don't forget to run it!

Looking Things Over

Open the diagram and add the new tables to the diagram. The diagram now looks something like the one shown in Figure 5-15.

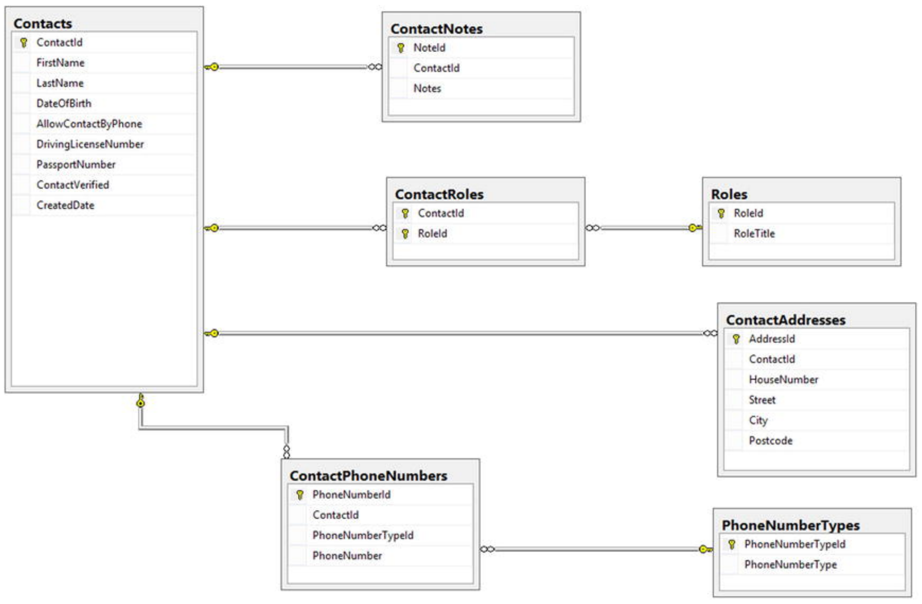


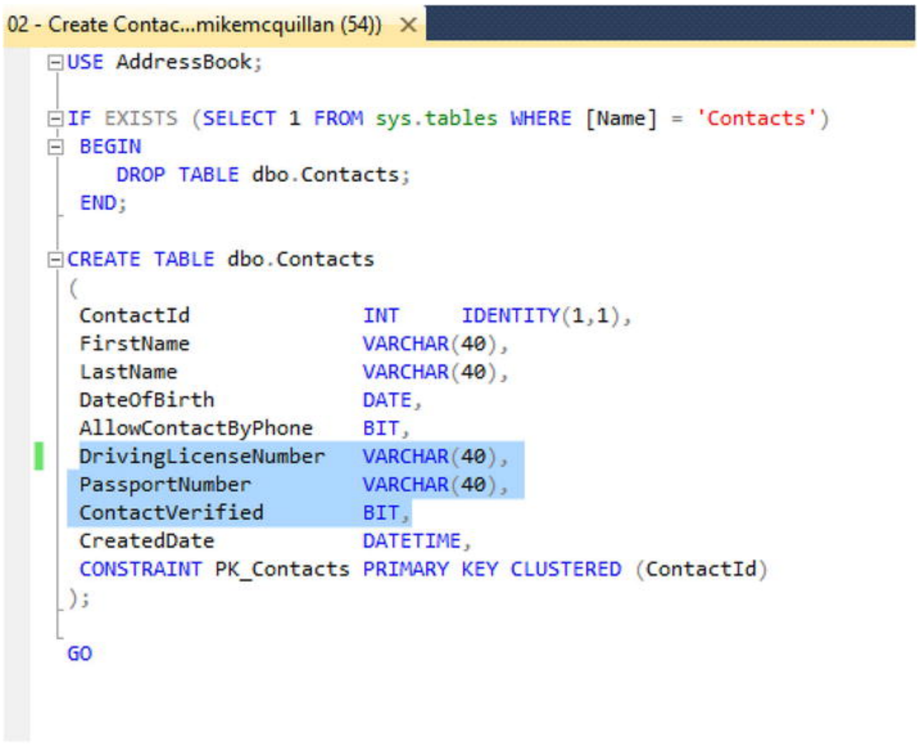
Figure 5-15. *Enhanced database diagram with improved phone numbers*

The `ContactPhoneNumbers` table looks like it is in the middle of a many-to-many relationship. It isn't. The `Contacts` table has a one-to-many relationship with `ContactPhoneNumbers`, via the `ContactId` column. The `PhoneNumberTypes` table has a one-to-many relationship with `ContactPhoneNumbers` via the `PhoneNumberTypeId` column. It is only a many-to-many relationship if the same column is used in all sides of the relationship.

A Second Enhancement

I mentioned there were one or two enhancements we could make, but so far we've only made one change. Was I telling a fib? Of course not. We can split up our `Contacts` table into a one-to-one relationship with another table, so we can logically group all of our verification information together. We don't know if the number of verification fields will grow in the future. If they do, having them in their own table will make a lot of sense.

We'll begin by opening `c:\temp\sqlbasics\apply\02 - Create Contacts Table.sql`. Highlight the lines from `DrivingLicenseNumber` to `ContactVerified`, then **cut** them using CTRL-X on your keyboard or the **Edit ➤ Cut** menu option. You can see the highlighted columns in Figure 5-16.



02 - Create Contac...mikemcquillan (54))

```
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,
 AllowContactByPhone     BIT,
 DrivingLicenseNumber    VARCHAR(40),
 PassportNumber          VARCHAR(40),
 ContactVerified         BIT,
 CreatedDate             DATETIME,
 CONSTRAINT PK_Contacts PRIMARY KEY CLUSTERED (ContactId)
);

GO
```

Figure 5-16. *Highlighting columns for cutting*

The `CREATE TABLE` statement should be left looking like this:

```
CREATE TABLE dbo.Contacts
(
ContactId INT IDENTITY(1,1),
FirstName VARCHAR(40),
LastName VARCHAR(40),
DateOfBirth DATE,
AllowContactByPhone BIT,
CreatedDate DATETIME,
CONSTRAINT PK_Contacts PRIMARY KEY CLUSTERED (ContactId)
);
```

Open up a New Query Window and type the following script to create the `ContactVerificationDetails` table. You can paste most of the columns in.

```
USE AddressBook;

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

CREATE TABLE dbo.ContactVerificationDetails
(
ContactId INT,
DrivingLicenseNumber VARCHAR(40),
PassportNumber VARCHAR(40),
ContactVerified BIT,
CONSTRAINT PK_ContactVerificationDetails PRIMARY KEY CLUSTERED (ContactId)
);

ALTER TABLE dbo.ContactVerificationDetails
ADD CONSTRAINT FK_ContactVerificationDetails_Contacts
FOREIGN KEY (ContactId)
REFERENCES dbo.Contacts (ContactId)
ON UPDATE NO ACTION
ON DELETE CASCADE;

GO
```

Nothing we haven't seen before. Note that we declare the `ContactId` as the primary key, which means we'll have a one-to-one relationship to the `Contacts` table.

Run this script to create the table, and save it as `c:\temp\sqlbasics\09 - Create ContactVerificationDetails Table.sql`.

Open a New Query Window and execute this statement to remove the transferred columns from the `Contacts` table:

```
ALTER TABLE dbo.Contacts
DROP COLUMN DrivingLicenseNumber, PassportNumber, ContactVerified;
```

Close this script without saving it. Now open our `Contacts` diagram and add the `ContactVerificationDetails` table to it. Figure 5-17 shows our completed diagram, with the entire database structure.

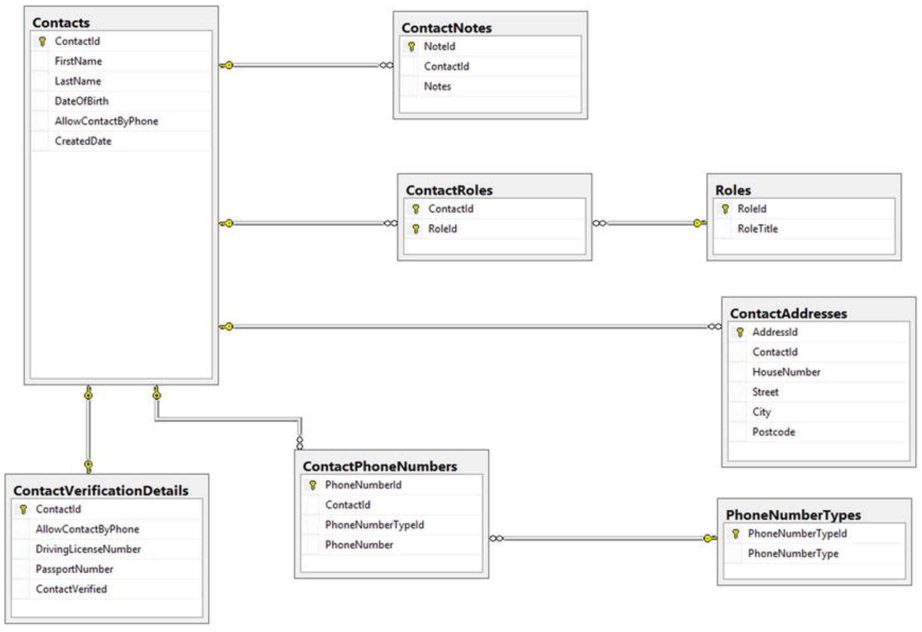


Figure 5-17. *Completed database diagram*

Summary

Well, we've covered a huge amount of ground in the last two chapters. Hopefully, you can now see how vital it is that you correctly structure the tables and how important it is that you take some time to think about how you want your data to be stored.

We're still not quite done with tables, although it's fair to say we've completed the heavy lifting. We've created a bunch of scripts. Now we need to figure out how we can easily execute them in one go. We also need to create some rollback scripts, which will allow us to put the server back to how it was, should we need to.

PREV
Chapter 4 : Tables

NEXT
Chapter 6 : Automating Deploym..