CHAPTER 7

NULLs and Table Constraints

You've done brilliantly. You've created an entire database with a normalized table structure. You've even created specialized scripts so you can build or drop the database easily. Now we're going to look at how we can help to ensure—as far as we can—that only good data is entered into our tables. As you might imagine, SQL Server provides a few tools that can help. We're going to look at the following:

- `NULLS`: what they are, and how we can specify whether a column supports them or not
- **Default Constraints**: why and how to use them
- **Table Constraints**: You've actually already met some of these, when we created primary keys and foreign keys in Chapter 4 There are two other types we'll look at: `CHECK` and `UNIQUE`.

NULL Constraints

I've mentioned `NULL` a few times in this book now. But just what is `NULL`? `NULL` is nothing. Nada. Zip. `NULL` means no value has been provided for a particular column. Look at this record in Figure 7-1.

BERTIE1\SQLEXPRES...ok - dbo.Contacts

	ContactId	FirstName	LastName
	1	Mike	*NULL*

Figure 7-1. *A record with a `NULL LastName` value*

We have a record here with a `FirstName`, but no `LastName`. It's important to understand that this is very different from the record in Figure 7-2.

BERTIE1\SQLEXPRES...ok - dbo.Contacts

	ContactId	FirstName	LastName
	1	Mike	

Figure 7-2. *A record with an empty-string `LastName` value*

This record has a blank `LastName`. `NULL` means no value has been provided for `LastName`; blank means an empty string has been provided. Let's see how these values affect our queries. If you want to find records for which a `LastName` has *not* been specified, use the special `IS` keyword, as the script in Figure 7-3 demonstrates.

SQLQuery12.sql - B...ikemcquillan (51))*

```
USE AddressBook;

SELECT ContactId, FirstName, LastName
    FROM dbo.Contacts
WHERE LastName IS NULL;
```

Figure 7-3. *Filtering for `NULL` values using `IS NULL`*

As Figure 7-4 shows, we can simply use the = operator if a value has been specified, such as an empty string.

SQLQuery12.sql - B...ikemcquillan (51))*

```
USE AddressBook;

SELECT ContactId, FirstName, LastName
    FROM dbo.Contacts
WHERE LastName IS NULL;

SELECT ContactId, FirstName, LastName
    FROM dbo.Contacts
WHERE LastName = ''
```

100 %

Results | Messages

	ContactId	FirstName	LastName

	ContactId	FirstName	LastName
1	1	Mike	

Figure 7-4. *Filtering for empty string values using =*

The first query, which returned a record before, now returns nothing. The second query, matching the empty string, successfully returns the record. When searching for `NULL` values, you must use `IS NULL` or `IS NOT NULL`—you cannot use `= NULL` or `!= NULL`. `NULL` is a special value and is treated differently from all other values. Be aware that if you want to change a string column value in SSMS to `NULL`, you must either type `NULL` in uppercase or press Ctrl+0 (make sure you select the column first).

NULL = NULL

You might think `NULL = NULL` would return `TRUE`. It doesn't, because `NULL` means an indistinct value. So `NULL` never equals `NULL`. The only way you can make `NULL = NULL` at the moment is to run the command `SET ANSI_NULLS OFF`, which will turn off ISO-compliant treatment of `NULL` in SQL Server. This functionality is deprecated and will be disabled in a future version of SQL Server, so using it is not advised. Also, `NULL` doesn't equal `NULL`, so you shouldn't use it anyway!

Don't worry about the details of these queries—you'll be introduced to them soon enough. For now, just remember that `NULL` means no value has been provided. In a `Contacts` table I don't think it's a good idea to allow `NULL` values for the `FirstName` and `LastName` columns—surely every contact has a name? Let's quickly revisit the `CREATE TABLE` statement for the `Contacts` table:

```
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)
);
```

What we want to do here is force the user to supply `FirstName` and `LastName` values. We don't necessarily need a `DateOfBirth` value, so we'll allow `NULL`s for that. We probably do want values to be supplied for the other columns. Open up script `02` and change the `CREATE TABLE` statement so it looks like this:

```
CREATE TABLE dbo.Contacts
(
ContactId INT IDENTITY(1,1) NOT NULL,
FirstName VARCHAR(40) NOT NULL,
LastName VARCHAR(40) NOT NULL,
DateOfBirth DATE NULL,
AllowContactByPhone BIT NOT NULL,
CreatedDate DATETIME NOT NULL,
CONSTRAINT PK_Contacts PRIMARY KEY CLUSTERED (ContactId)
);
```

We've added either `NULL` or `NOT NULL` to the end of each column declaration. The `DateOfBirth` column has `NULL` next to it, denoting `NULL` values are allowed. The other columns all specify `NOT NULL`, which means a value must be supplied when a row is being inserted into the table.

Save the script. Then run the `00 - Rollback.sql` script in the rollback folder, and then run the `00 - Apply.sql` file in the apply folder (don't forget to turn on SQLCMD mode before running these). The database will be recreated, along with the `NULL` constraints on the `Contacts` table. Refresh the **AddressBook** database in the Object Explorer, find the `Contacts` table, and open it for editing (right-click the `Contacts` table and choose **Edit Top 200 Rows**).

Enter a name into the `FirstName` column, and then press Enter. Lo and behold, the error message in Figure 7-5 is displayed.

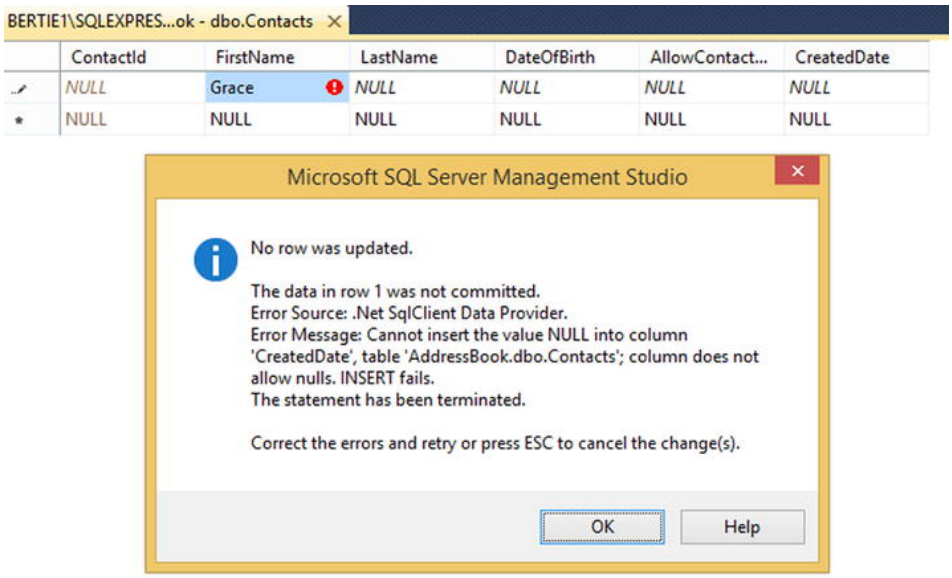


Figure 7-5. *`NOT NULL` constraint preventing a new row insert*

This is interesting—SQL Server would not let us add our new row. The error message states it's because no value was provided for the `CreatedDate` column. We'll keep seeing these errors until all columns that do not allow NULLs have been populated. If you populate all columns except `DateOfBirth` you should be allowed to add your row. The `AllowContactByPhone` column can accept `TRUE`, `FALSE`, `1` (for `TRUE`), or `0` (for `FALSE`). The `CreatedDate` column requires a date, in the format `YYYY-MM-DD`. You can add a time, too if you want, in the format `YYYY-MM-DD HH:MM:SS`. Once all values have been correctly specified you'll see a new row, just like the one in Figure 7-6.

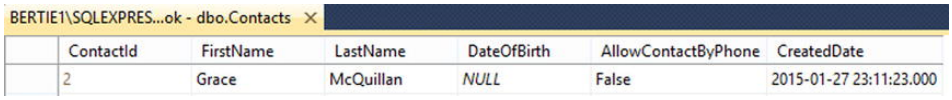


Figure 7-6. *Adding a new row after creating `NULL` constraints*

That's all there is to `NULL` constraints. We didn't need to specify `NULL` on the `ContactId` column, as it's a primary key and is therefore required anyway. But as I've said many times already, always be explicit so it's obvious what you are trying to do.

We need to add appropriate `NULL` constraints for the other tables in our database. Modify the apply scripts as directed (no need to touch the rollback scripts).

- `03 - Create ContactNotes Table.sql`

  All columns should be `NOT NULL`—there isn't much point in a `NULL` note.

```
CREATE TABLE dbo.ContactNotes
(
NoteId INT IDENTITY(1,1) NOT NULL,
ContactId INT NOT NULL,
Notes VARCHAR(200) NOT NULL,
CONSTRAINT PK_ContactNotes PRIMARY KEY CLUSTERED (NoteId)
);
```

- `04 - Create Roles Table.sql`

  Again, make all columns `NOT NULL`. A role must have a title and an ID.

```
CREATE TABLE dbo.Roles
(
RoleId INT IDENTITY(1,1) NOT NULL,
RoleTitle VARCHAR(200) NOT NULL,
CONSTRAINT PK_Roles PRIMARY KEY CLUSTERED (RoleId)
);
```

- `05 - Create ContactRoles Table.sql`

  Yet again, all columns should be `NOT NULL`. The many-to-many link record is useless without both values.

```
CREATE TABLE dbo.ContactRoles
(
ContactId INT NOT NULL,
RoleId INT NOT NULL,
CONSTRAINT PK_ContactRoles PRIMARY KEY CLUSTERED (ContactId, RoleId)
);
```

- `06 - Create ContactAddresses Table.sql`

  It's difficult to determine the values a user may provide for an address, so we'll make all of the address columns optional. Just the keys are required.

```
CREATE TABLE dbo.ContactAddresses
(
AddressId INT IDENTITY(1,1) NOT NULL,
ContactId INT NOT NULL,
HouseNumber VARCHAR(200) NULL,
Street VARCHAR(200) NULL,
City VARCHAR(200) NULL,
Postcode VARCHAR(20) NULL,
CONSTRAINT PK_ContactAddresses PRIMARY KEY NONCLUSTERED (AddressId)
);
```

- `07 - Create PhoneNumberTypes Table.sql`

  There's no point in having a `NULL` phone number type, so all columns here are required.

```
CREATE TABLE dbo.PhoneNumberTypes
(
PhoneNumberTypeId TINYINT IDENTITY(1,1) NOT NULL,
PhoneNumberType VARCHAR(40) NOT NULL,
CONSTRAINT PK_PhoneNumberTypes PRIMARY KEY CLUSTERED (PhoneNumberTypeId)
);
```

- `08 - Create ContactPhoneNumbers Table.sql`

  Again, a phone number record is useless without a phone number! We want all columns populated.

```
CREATE TABLE dbo.ContactPhoneNumbers
(
PhoneNumberId INT IDENTITY(1,1) NOT NULL,
ContactId INT NOT NULL,
PhoneNumberTypeId TINYINT NOT NULL,
PhoneNumber VARCHAR(30) NOT NULL
CONSTRAINT PK_ContactPhoneNumbers PRIMARY KEY CLUSTERED (PhoneNumberId)
);
```

- `09 - Create ContactVerificationDetails Table.sql`

  Our final table, and this has some optional values. We don't know what the user will enter for verification purposes, so we'll make `DrivingLicenseNumber` and `PassportNumber` both optional.

```
CREATE TABLE dbo.ContactVerificationDetails
(
ContactId INT NOT NULL,
DrivingLicenseNumber VARCHAR(40) NULL,
PassportNumber VARCHAR(40) NULL,
ContactVerified BIT NOT NULL
CONSTRAINT PK_ContactVerificationDetails PRIMARY KEY CLUSTERED (ContactId));
```

Excellent! Close all open windows and roll back the database (execute the `00 - Rollback.sql` script). In certain circumstances this may fail—SSMS keeps a background process connected to the database, preventing it from being dropped. If you see an error saying **Cannot drop database AddressBook because it is currently in use**, you need to close SSMS, reopen it, open the rollback script, and then execute it again. This time it will work fine.

Once you've successfully rolled back, open and run `00 - Apply.sql`. The database is now locked down to prevent `NULL` values where necessary. Do you see how easy SQL Server makes it for you to tighten up data entry?

And we're not finished yet!

Constraints

We are now going to take a look at two types of constraint: *default constraints* and *table constraints*. Both are useful, as you'll see.

Default Constraints

A default constraint is assigned to a column when a new row is inserted into a table and no value has been specified for that column. Say the `CreatedDate` column in the `Contacts` table had a specific date assigned to it as a default value. Inserting a new row would cause that date to be set as the `CreatedDate` column value, unless an alternative value was specified.

You shouldn't assign default values to every column; just where it makes sense. Defaults usually work well for `BIT` columns (defaulting them to `TRUE` or `FALSE`), and columns like `CreatedDate`, which are generally used for auditing purposes.

There are only two tables we want to assign default values for in our database: `Contacts` and `ContactVerificationDetails`. We'll assign the following defaults:

- `Contacts`: set a default of `FALSE` on `AllowContactByPhone`
- `Contacts`: set a default of the current date/time on `CreatedDate`
- `ContactVerificationDetails`: set a default of `FALSE` on ContactVerified

Note that none of these columns allow `NULL` values, which is another good reason for using default values (bear in mind that a `NULL` value can be manually supplied via an `INSERT` statement).

Creating a Default Constraint

Open up `c:\temp\sqlbasics\apply\02 - Create Contacts Table.sql`. Find the `AllowContactByPhone` line and change it by adding a `DEFAULT` clause:

```
AllowContactByPhone BIT NOT NULL DEFAULT 0,
```

That's all you need to do—the `0` means `FALSE`. Rebuild the database (run the `00 - Rollback.sql` script, then the `00 - Apply.sql` script—whenever I mention rebuilding the database going forward, this is what you should do) and then add a new record to the `Contacts` table (using the **Edit Top 200 Rows** option), *without* specifying a value for the `AllowContactByPhone` column. Before you press Enter, you should see something like Figure 7-7.

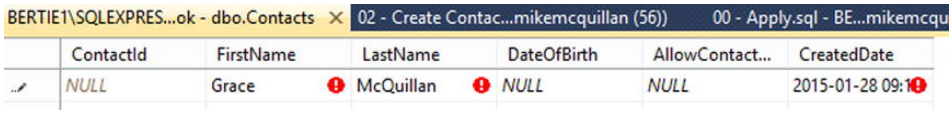


Figure 7-7. *About to commit a new row with default values*

Now press Enter to commit the row. The row will change—`AllowContactByPhone` still displays `NULL`, but there's a red exclamation mark to the left of the row (you can see this in Figure 7-8). This indicates we are not looking at the current version of the row.

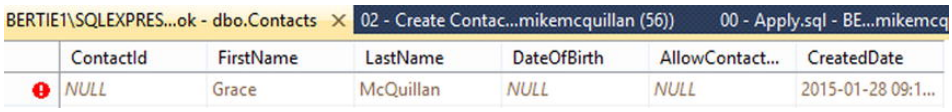


Figure 7-8. *A newly committed row that needs refreshing*

Press Ctrl+R on your keyboard to refresh the table's contents. The red exclamation mark should disappear, the `ContactId` column should be populated, and the `AllowContactByPhone` column should change to `FALSE` (Figure 7-9).

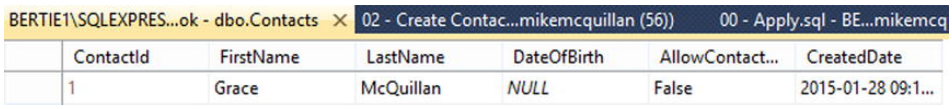


Figure 7-9. *The new row, refreshed with default values*

Great, isn't it? But it's not a bed of roses just yet. In the Object Explorer, expand the `Contacts` table (**Databases** ➤ **AddressBook** ➤ **Tables** ➤ **Contacts**), and expand **Constraints**. Uh-oh—as Figure 7-10 shows, things don't look pleasant.

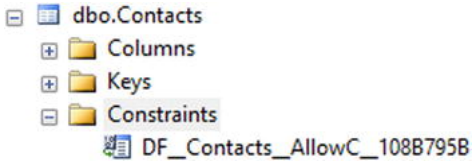


Figure 7-10. *A badly named default constraint*

Can you remember when we first created a primary key? SQL Server appended a weird combination of letters and numbers to the end of the primary key name, making it difficult to maintain should we need to amend it in the future. A similar thing has happened here. We didn't provide a name for the constraint, so SQL Server has provided one for us. This name is of no use. We'd never remember it, and it isn't explicit—we want to stay in control of the names of our objects. So we need to return to our `02 - Create Contacts Table.sql` script. Change the `AllowContactByPhone` line:

```
AllowContactByPhone BIT NOT NULL CONSTRAINT DF_Contacts_AllowContactByPhone DEFAULT 0,
```

Rebuild the database and return to the `Contacts` table's **Constraints** node in the Object Explorer. Refresh this; it should now be showing us the name we just specified, as in Figure 7-11.

Figure 7-11. *A well-named default constraint*

Much better! For minimal effort we've added a name that makes sense, which makes things easier for us to manage in the future.

We can now go ahead and modify the `CreatedDate` column in the same script. This is a value used purely for auditing, so there is need to ask the person creating the record to supply it. Change the `CreatedDate` line:

```
CreatedDate DATETIME NOT NULL CONSTRAINT DF_Contacts_CreatedDate DEFAULT GETDATE()
```

This looks very similar to the `AllowContactByPhone` default constraint, except we specify `GETDATE()` as the value (you always specify the value you want as the default after the `DEFAULT` keyword). `GETDATE()` is one of SQL Server's system functions (there's a list of some of the more useful functions in Appendix C). It returns the current date and time.

Save the `02 - Create Contacts Table.sql` script. Before we rebuild the database, we may as well modify the `09 - Create ContactVerificationDetails Table.sql` script. We want to specify a default value on the `ContactVerified` column.

```
ContactVerified BIT NOT NULL CONSTRAINT
DF_ContactVerificationDetails_ContactVerified DEFAULT 0
```

Save this script, too, and then rebuild the database. Try adding a new record to `Contacts`, but specify just a `FirstName` and `LastName` (as I've done in Figure 7-12).

BERTIE1\SQLEXPRES...ok - dbo.Contacts | 00 - Apply.sql - BE...mikemcquillan (55)) | 00 - Rollback.sql -...\mikemcc

	ContactId	FirstName	LastName	DateOfBirth	AllowContact...	CreatedDate
	NULL	Grace	McQuillan	*NULL*	*NULL*	*NULL*

Figure 7-12. *Creating a new row with default constraints*

Press Enter, then Ctrl+R to refresh the rows. As Figure 7-13 demonstrates, you should be left with just a `NULL DateOfBirth`.

BERTIE1\SQLEXPRES...ok - dbo.Contacts | 00 - Apply.sql - BE...mikemcquillan (55)) | 00 - Rollback.sql -...\mikemcquillan (5

	ContactId	FirstName	LastName	DateOfBirth	AllowContact...	CreatedDate
	1	Grace	McQuillan	*NULL*	False	2015-01-28 09:54:29.980

Figure 7-13. *A new row displaying default values*

We're all done with default values. Just remember that defaults are only applied when you create a row, not when you update it. But they are very useful, especially when used in conjunction with stored procedures, as we'll see later.

Table Constraints Part 1—Check Constraints

A Check constraint is very different from a default, and it will be applied to new rows and existing rows that are being updated. Use a Check constraint to ensure valid values are entered into your rows—the insert or update operation is rejected if the Check constraint is not met.

When we were looking at `NULL` earlier, you may remember we added a record with a blank `LastName` value. This isn't a `NULL` value, but rather an empty string. Try it—you can still do this (Figure 7-14 shows the proof!).

BERTIE1\SQLEXPRES...ok - dbo.Contacts

	ContactId	FirstName	LastName
	1	Mike	

Figure 7-14. *Specifying an empty `LastName` value*

You can also enter an empty `FirstName` should you wish. This is not good—we'll use a Check constraint to solve this problem. The rules for Check constraints are the following:

- Check constraints can be applied to multiple columns (by specifying them at the table level), or to just one column (by specifying them at the column level).
- The expression you specify must return `TRUE` or `FALSE`.
- Multiple Check constraints can be applied to a single column.
- Any valid T-SQL expression can be used as part of a Check constraint, as long as it returns `TRUE` or `FALSE`.

If the Check constraint returns `FALSE`, the constraint has failed and the insert or update operation is rejected. Because of this, you need to ensure any programs you or other developers may write to insert or update data are aware that an error may occur should the constraints not be met. Of course, preventing bad data and allowing systems to do something logical when bad data is provided is the main reason why we implement Check constraints.

We are going to add the following Check constraints to our database:

- `Contacts`: Ensure that valid `FirstName` and `LastName` values are provided.
- `Contacts`: Limit `DateOfBirth` to values greater than January 1, 1850.
- `ContactAddresses`: Ensure a valid value is provided for at least one of `HouseNumber`, `Street`, `City`, or `Postcode`.

Creating a Table Constraint

Because the first and third constraints I mentioned involve multiple columns, these constraints will be created at the table level. The second constraint, to limit `DateOfBirth`, will be a column constraint.

Open the `02 - Create Contacts Table.sql` script from the apply folder. Add a new line to the bottom of the `CREATE TABLE` statement.

```
CREATE TABLE dbo.Contacts
(
ContactId INT IDENTITY(1,1) NOT NULL,
FirstName VARCHAR(40) NOT NULL,
LastName VARCHAR(40) NOT NULL,
DateOfBirth DATE NULL,
AllowContactByPhone BIT NOT NULL CONSTRAINT DF_Contacts_AllowContactByPhone DEFAULT 0,
CreatedDate DATETIME NOT NULL CONSTRAINT DF_Contacts_CreatedDate DEFAULT GETDATE(),
CONSTRAINT PK_Contacts PRIMARY KEY CLUSTERED (ContactId),
CONSTRAINT CK_Contacts_FirstNameLastName CHECK (FirstName != '' OR LastName != '')
);
```

We give the Check constraint a name, beginning with `CK` for Check—yes, sensible naming strikes again! Then we have the keyword `CHECK`, followed by some text wrapped up in brackets. The text within the brackets is your *check expression*. This is what SQL Server will evaluate whenever a value is provided for the `FirstName` or `LastName` columns. You can use virtually any valid T-SQL expression in a Check constraint.

Save the script and rebuild your database. If you'd rather not rebuild the database, you can run the following `ALTER TABLE` statement:

```
ALTER TABLE dbo.Contacts
ADD CONSTRAINT CK_Contacts_FirstNameLastName CHECK (FirstName != '' OR LastName != '');
```

Note the name clearly describes the table and columns involved with the constraint. If you have multiple constraints for the same columns, consider adding an extra piece of information to the end of the name.

REBUILDING VS. ALTERING

We've been consistently rebuilding the database throughout this chapter. This is fine, as our database doesn't contain any data yet. Once a database is put into production it will contain data, and rebuilding it ceases to be an option.

When this happens, you will need to write scripts that `ALTER` existing objects, rather than `CREATE` them.

If you ran the `ALTER TABLE` statement, you'll see the now familiar **Command(s) completed successfully** message. Let's go ahead and test our constraint.

Open up the `Contacts` table for editing and add a row. Make sure you enter a blank value for the `FirstName` column (just press Delete to clear out the `NULL`), and a valid value for the `LastName` column. Your row should look something like the second row in Figure 7-15.

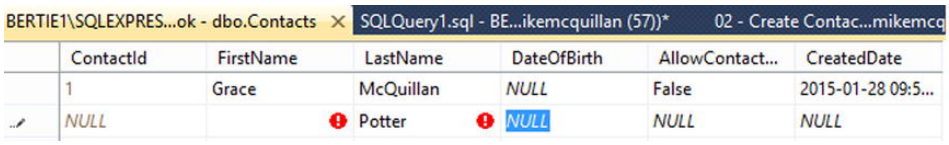


Figure 7-15. *About to commit a new row with a blank `FirstName` value*

Once you are ready, press Enter—all other columns have defaults or allow `NULL`s. We are expecting to see an error message appear, rejecting the row.

What actually happens is a red exclamation mark appears to the left of the row, shown in Figure 7-16.

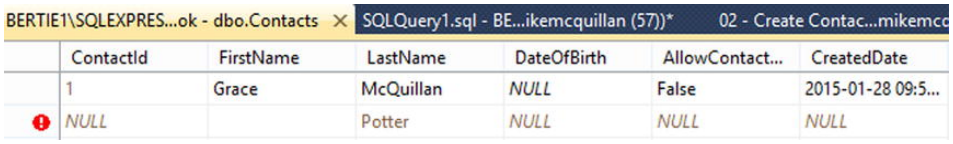


Figure 7-16. *A committed new row awaiting a refresh*

We saw this earlier. Press Ctrl+R to refresh the table. As Figure 7-17 shows, the row has been accepted!

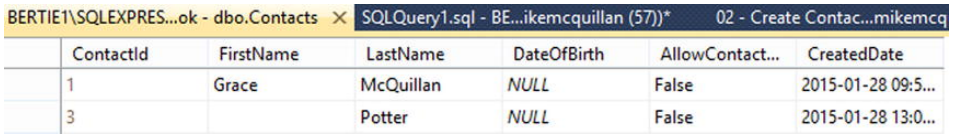


Figure 7-17. *A refreshed row with an empty `FirstName` value*

This isn't good. We've (well, I've) done something wrong. Just as a test, blank out the `LastName` column, too, and press Enter. Do you see the error message in Figure 7-18?

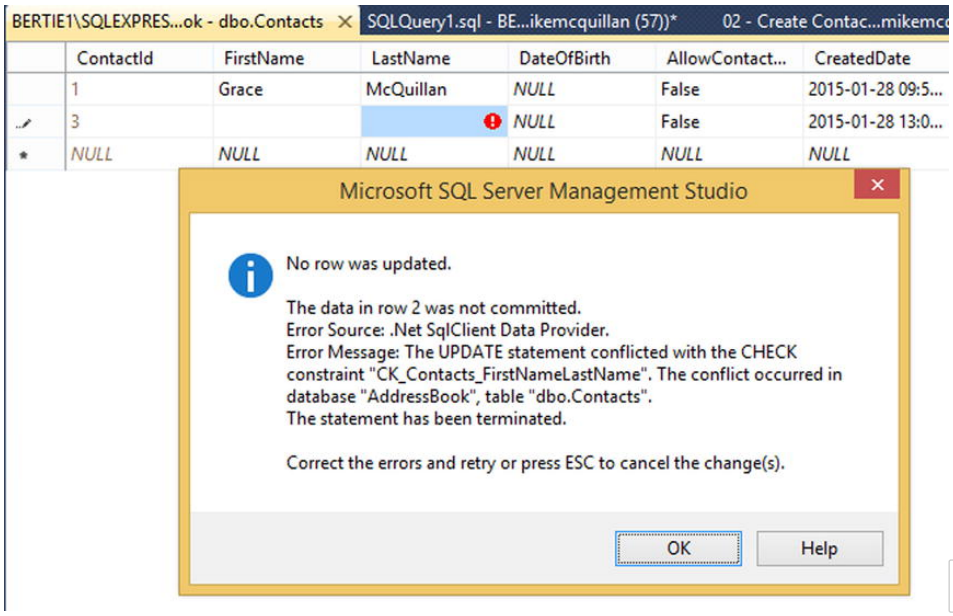


Figure 7-18. *Attempting to update a row with blank* `FirstName` *and* `LastName` *entries*

Ah-ha! The error message in **Figure 7-18** is what we were expecting to see earlier, an error telling us there is a conflict with the `CK_Contacts_FirstNameLastName` constraint. So things work if we leave both values empty. What the heck is going on?

I'll analyze the constraint expression. `FirstName != ''` checks if an empty string has been provided for `FirstName`. != means `NOT EQUAL TO`. So if `FirstName` is Mike, this will return `TRUE`, as Mike is `NOT EQUAL TO` an empty string. If `FirstName` is an empty string, this will return `FALSE`.

The check for `LastName` is exactly the same. They are joined by the keyword `OR`. This means only one of these conditions has to be met. So if `FirstName` is an empty string but `LastName` isn't, the constraint has been met. And vice versa—if we have a `FirstName` but no `LastName`, we have one of the values, so the constraint has been met.

This isn't what we want—we want the constraint to ensure both values have been provided. Let's drop the constraint and change it to use `AND` instead. Open a New Query Window and type in this code:

```
ALTER TABLE dbo.Contacts DROP CONSTRAINT CK_Contacts_FirstNameLastName;

ALTER TABLE dbo.Contacts ADD CONSTRAINT CK_Contacts_FirstNameLastName CHECK (FirstName != '' AND LastName != '');
```

This is a subtle but important change. This expression is saying both `FirstName` and `LastName` must not be empty strings for the insert or update to fail. So if one of them is empty, the check will not be met.

Run this. If you have a row in the table with either a blank `FirstName` or `LastName` column value, the constraint won't be created—you'll see a red warning message, just like the one in Figure 7-19.

Figure 7-19. *A failed attempt to create a Check constraint*

This is because the data currently in the table don't meet the constraint. This is good; SQL Server is telling us to clean up our data before creating the constraint. To delete the offending row, return to the `Contacts` table in the table editor, right-click the gray box to the left of the row, and choose the **Delete** option as shown in Figure 7-20.

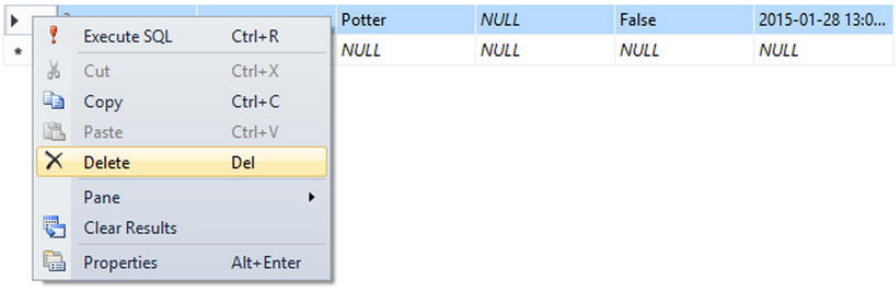


Figure 7-20. *Deleting the noncompliant row*

You will be asked to confirm the deletion. Do so, then return to the query window containing the constraint code.

CREATING CHECK CONSTRAINTS WHEN NONCOMPLIANT DATA EXISTS

Often, you'll want to introduce new Check constraints after a system has been live for a while. The table you are adding the constraint to may contain rows that don't meet the new Check constraint, and editing or deleting them may not be an option. Fear not—you can use the `WITH CHECK` and `WITH NOCHECK` options when creating the constraint. `WITH CHECK` is the default and will cause data to be validated against the new constraint. If you don't want to validate the existing data, use `WITH NOCHECK`. The constraint will only apply to new and updated rows.

These clauses apply to foreign keys too.

Highlight the `ALTER TABLE ADD CONSTRAINT` code (like I've done in Figure 7-21) and then press F5. This is a very nice feature of SSMS; if you have highlighted some code, it will only execute that code. We don't want to execute the `ALTER TABLE DROP CONSTRAINT` code—that executed earlier when we first tried to recreate the constraint. The `DROP CONSTRAINT` succeeded, but the `ADD CONSTRAINT` failed because of the bad data in the table.

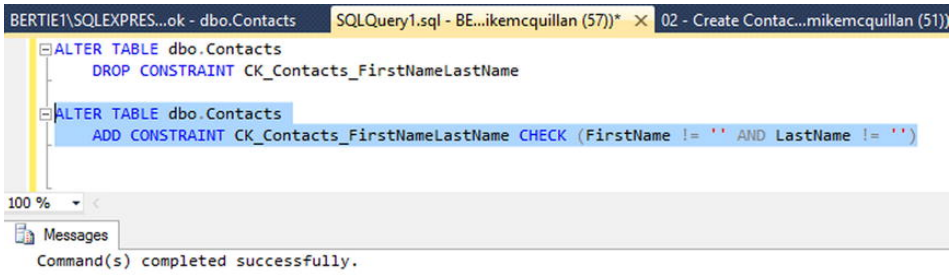
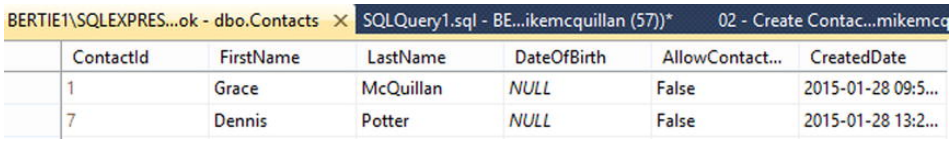


Figure 7-21. *Creating a modified Check constraint*

Now return to the table editor and try to add a new row, with a blank `FirstName` and a valid `LastName`. Press Enter to commit the row. What happens? The error message appears, telling us the constraint has not been met. Hurrah! Click **OK** on the error message and blank out the `LastName`, too. Press Enter again. You should still see the error message. Now try specifying a valid `FirstName`, but no `LastName`. Press Enter, and again you should see the error message. Everything seems fine. Finally, enter a valid `FirstName` and `LastName`. The row should be accepted this time. Press Ctrl+R to refresh the table. You should see the two rows in Figure 7-22.


BERTIE1\SQLEXPRES...ok - dbo.Contacts | SQLQuery1.sql - BE...ikemcquillan (57))* | 02 - Create Contac...mikemcq

ContactId	FirstName	LastName	DateOfBirth	AllowContact...	CreatedDate
1	Grace	McQuillan	NULL	False	2015-01-28 09:5...
7	Dennis	Potter	NULL	False	2015-01-28 13:2...

Figure 7-22. *Inserting a valid row*

Great, our constraint is working! This is a pretty simple constraint we've added, but it serves to illustrate how powerful Check constraints can be.

Creating a Column Constraint

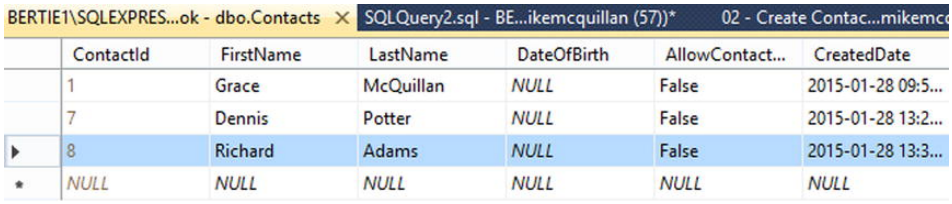
Make sure you amend the `02 - Create Contacts Table.sql` script with the change we made to the `CK_Contacts_FirstNameLastName` constraint. While you're there, add a new constraint at the bottom to check if `DateOfBirth` values are greater than January 1, 1850. Here's the full `CREATE TABLE` statement:

```
CREATE TABLE dbo.Contacts
(
ContactId INT IDENTITY(1,1) NOT NULL,
FirstName VARCHAR(40) NOT NULL,
LastName VARCHAR(40) NOT NULL,
DateOfBirth DATE NULL,
AllowContactByPhone BIT NOT NULL CONSTRAINT DF_Contacts_AllowContactByPhone DEFAULT 0,
CreatedDate DATETIME NOT NULL CONSTRAINT DF_Contacts_CreatedDate DEFAULT GETDATE(),
CONSTRAINT PK_Contacts PRIMARY KEY CLUSTERED (ContactId),
CONSTRAINT CK_Contacts_FirstNameLastName CHECK (FirstName != '' AND LastName != ''),
CONSTRAINT CK_Contacts_DateOfBirth CHECK (DateOfBirth > '1850-01-01')
);
```

You can either rebuild the database, or run this `ALTER TABLE` statement:

```
ALTER TABLE dbo.Contacts ADD CONSTRAINT CK_Contacts_DateOfBirth CHECK (DateOfBirth > '1850-01-01');
```

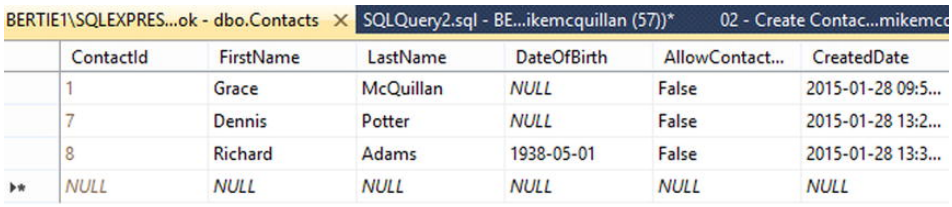
Return to the `Contacts` table's table editor and add a new row, leaving `NULL` as the `DateOfBirth` value. The row should be created as normal, even though we didn't specify a `DateOfBirth`. You can see it in Figure 7-23.


BERTIE1\SQLEXPRES...ok - dbo.Contacts | SQLQuery2.sql - BE...ikemcquillan (57))* | 02 - Create Contac...mikemcq

ContactId	FirstName	LastName	DateOfBirth	AllowContact...	CreatedDate
1	Grace	McQuillan	NULL	False	2015-01-28 09:5...
7	Dennis	Potter	NULL	False	2015-01-28 13:2...
8	Richard	Adams	NULL	False	2015-01-28 13:3...
NULL	NULL	NULL	NULL	NULL	NULL

Figure 7-23. *Inserting a new row without a* `DateOfBirth`

This is correct, as we have configured `DateOfBirth` to allow `NULL` values. Now, modify your new row—enter a `DateOfBirth` after January 1, 1850. This should be saved, too. Enter the date in the format `YYYY-MM-DD` (e.g., `1938-05-01` for May 1, 1938). Figure 7-24 shows the updated row.


BERTIE1\SQLEXPRES...ok - dbo.Contacts | SQLQuery2.sql - BE...ikemcquillan (57))* | 02 - Create Contac...mikemcq

ContactId	FirstName	LastName	DateOfBirth	AllowContact...	CreatedDate
1	Grace	McQuillan	NULL	False	2015-01-28 09:5...
7	Dennis	Potter	NULL	False	2015-01-28 13:2...
8	Richard	Adams	1938-05-01	False	2015-01-28 13:3...
NULL	NULL	NULL	NULL	NULL	NULL

Figure 7-24. *Updating the row with a* `DateOfBirth` *value*

Wonderful. We'll change that date to December 29, 1849 (`1849-12-29`). As expected, our Check constraint raises a complaint, which is displayed in Figure 7-25.

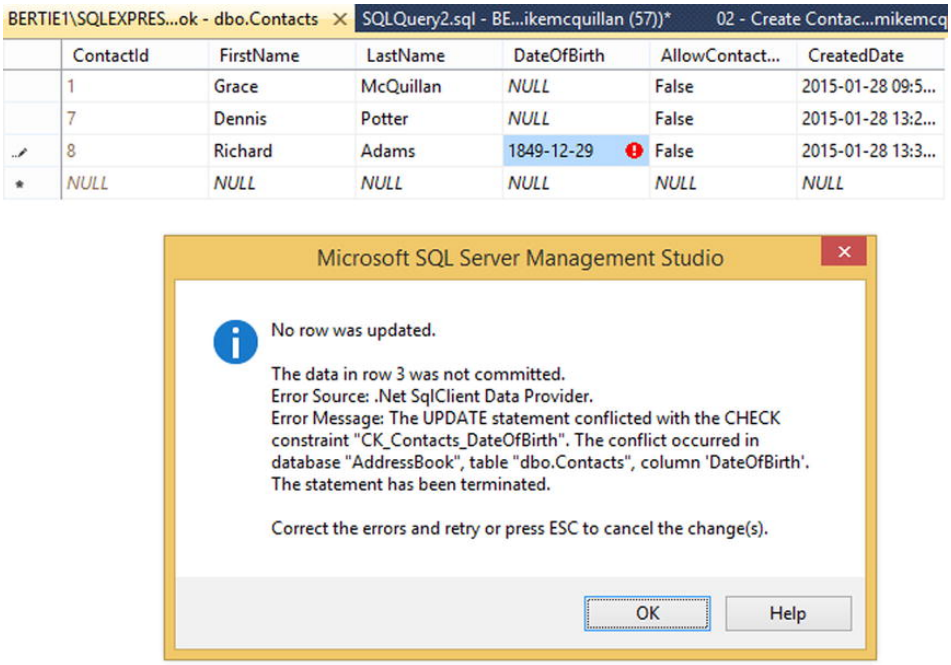


Figure 7-25. *Updating the row with an out-of-range* `DateOfBirth`

Click **OK** and press Esc to reject the changes. The `DateOfBirth` will revert back to 1938-05-01. Another Check constraint successfully created, this time as a column constraint. Now we only accept contacts who are roughly less than 165 years old!

One last example—we said we'd create a Check constraint on the `ContactAddresses` table, to ensure a valid value is provided for at least one of `HouseNumber`, `Street`, `City`, or `Postcode`. Note we only require one of these values, not all of them. We still use `AND` to check this. Open up `06 - Create ContactAddresses Table.sql` from the apply folder. Change the `CREATE TABLE` statement to include the Check constraint.

```
CREATE TABLE dbo.ContactAddresses
(
AddressId INT IDENTITY(1,1) NOT NULL,
ContactId INT NOT NULL,
HouseNumber VARCHAR(200) NULL,
Street VARCHAR(200) NULL,
City VARCHAR(200) NULL,
Postcode VARCHAR(20) NULL,
CONSTRAINT PK_ContactAddresses PRIMARY KEY NONCLUSTERED (AddressId),
CONSTRAINT CK_ContactAddresses_HouseNumberStreetCityPostcode
CHECK (HouseNumber != '' AND Street != '' AND City != '' AND Postcode != '')
);
```

Here's the `ALTER TABLE` statement so you don't need to rebuild your database:

```
ALTER TABLE dbo.ContactAddresses ADD CONSTRAINT
   CK_ContactAddresses_HouseNumberStreetCityPostcode CHECK
   (HouseNumber != '' AND Street != '' AND City != '' AND Postcode != '');
```

Make a note of a valid `ContactId` from the `Contacts` table, then open the table editor for the `ContactAddresses` table. Try adding addresses with all `NULL` values—this will be allowed, as will adding an address with at least one valid value specified (as long as the other values are not empty strings). But try adding an address with an empty string and it will be rejected, with the error message shown in **Figure 7-26**.

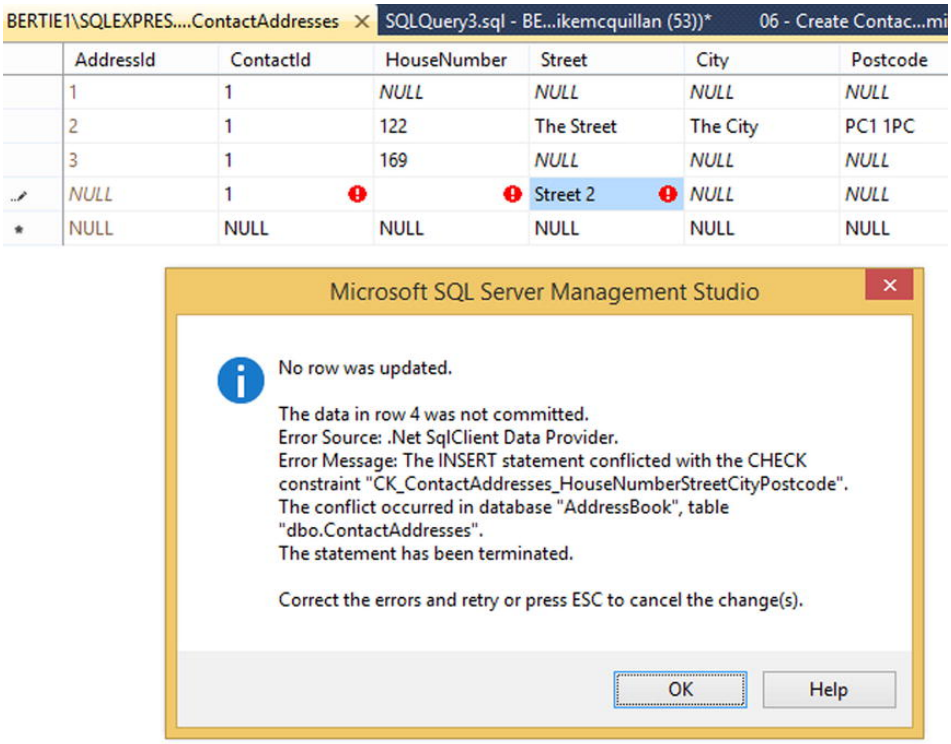


Figure 7-26. *Inserting an address without a* `HouseNumber` *value*

Congratulations, you've just made bad data entry much harder!

Table Constraints Part 2—Unique Constraints

To finish this chapter, we'll create a unique constraint. This is a special type of constraint that ensures a particular column, or combination of columns, in each row is unique. We'll add one unique constraint to the `ContactPhoneNumbers` table, which will ensure each `ContactId` and `PhoneNumber` combination is unique.

Open script `08 - Create ContactPhoneNumbers Table.sql` and modify the `CREATE TABLE` statement to include the `Unique` constraint.

```
CREATE TABLE dbo.ContactPhoneNumbers
(
PhoneNumberId INT IDENTITY(1,1) NOT NULL,
ContactId INT NOT NULL,
PhoneNumberTypeId TINYINT NOT NULL,
PhoneNumber VARCHAR(30) NOT NULL
CONSTRAINT PK_ContactPhoneNumbers PRIMARY KEY CLUSTERED (PhoneNumberId),
CONSTRAINT UQ_ContactIdPhoneNumber UNIQUE (ContactId, PhoneNumber)
);
```

This looks very similar to the constraints we've added already. It's more akin to a primary key constraint than a Check constraint. We give the constraint a name—beginning with `UQ` for `Unique` this time—and then declare the `UNIQUE` keyword, with the columns to check for uniqueness in brackets.

Here's the `ALTER TABLE` script for you to run:

```
ALTER TABLE dbo.ContactPhoneNumbers ADD CONSTRAINT
   UQ_ContactIdPhoneNumber UNIQUE (ContactId, PhoneNumber);
```

To test this, we must add a record to the `PhoneNumberTypes` table—the `PhoneNumberTypeId` column in the `ContactPhoneNumbers` table does not allow `NULL`s, and is a foreign key. Open the table editor for `PhoneNumberTypes` and add a `PhoneNumberType` of *Home*. Press Enter and make a note of the `PhoneNumberTypeId` given to this record. This value will probably be `1`, as you can see in Figure 7-27.

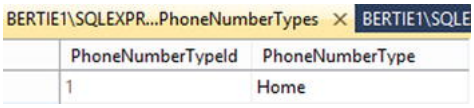


Figure 7-27. *Inserting a* `Home` `PhoneNumberType` *record*

Next, make sure you have two contacts present in the `Contacts` table. Make a note of their `ContactId`s and try adding phone numbers for them in the `ContactPhoneNumbers` table. For the same contact, try adding the same phone number two times. The second time, you'll see the error message shown in Figure 7-28.

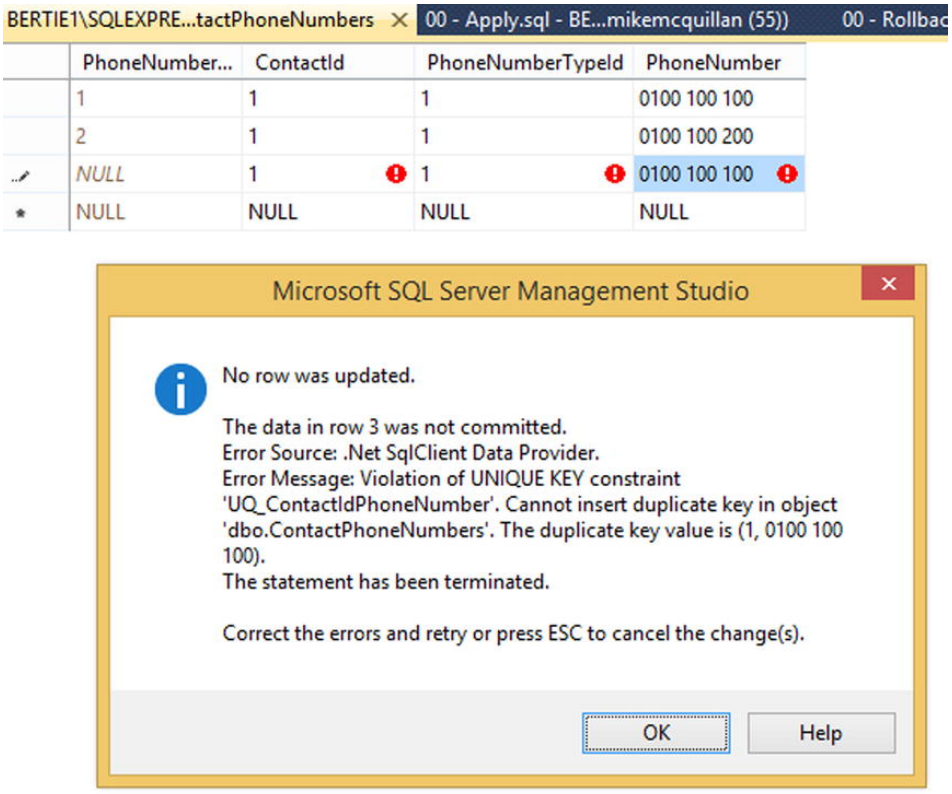


Figure 7-28. *A unique constraint in action*

The error message tells us that the unique constraint is working exactly as expected—the same contact cannot have the same phone number more than once. However, we can specify the same number for a different contact, as Figure 7-29 proves.

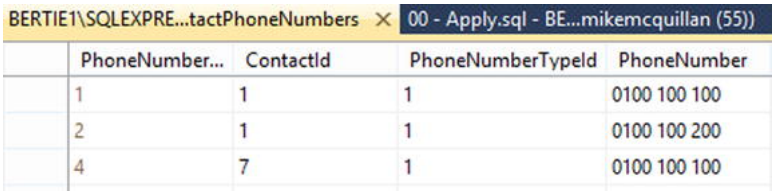


Figure 7-29. *Specifying the same phone number for two contacts*

This is correct, as the unique constraint applies to the combination of `ContactId` and `PhoneNumber`. We've now guaranteed that each contact will not have duplicate phone numbers.

Some Constraints Are Really Indexes

We haven't spoken about indexes yet—that will happen in Chapter 14. We've been merrily creating constraints in this chapter—or have we? Not quite. In the Object Explorer, expand **Databases** ➤ **AddressBook** ➤ **ContactPhoneNumbers** ➤ **Constraints**. Expand **Indexes**, too. There will be nothing under **Constraints**—eh?—but there will be some items beneath **Indexes**, which you can see in Figure 7-30.

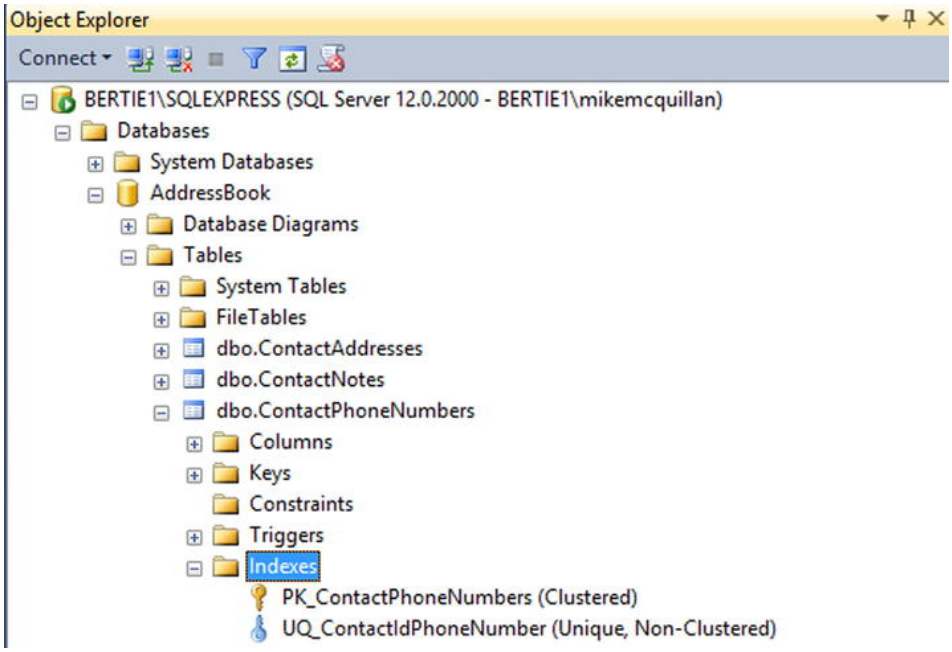


Figure 7-30. *Constraints that are actually indexes*

Both the primary key and the unique constraint have been created as indexes. This is because they *are* indexes! Check constraints and default constraints will be created as constraints, and will exist below the **Constraints** node. Primary keys are created as either a clustered or nonclustered index, and unique constraints are created as a unique index.

I mention this now just to avoid confusion should you go looking for your constraints! We'll talk about it in more detail later.

Summary

That was a marathon of a chapter. We've locked our database down pretty well. We started by determining which columns could accept `NULL` values, and then assigned some default values to certain columns. We continued to improve our database by adding some Check constraints, limiting the values supplied by users for certain columns. We finished off by introducing a unique constraint to the `ContactPhoneNumbers` table, preventing the same number from being provided multiple times for the same contact.

Everything has come together wonderfully, so much so that we've finished our table structure. Yes, our tables are now ready to be populated with some data—which means it's Data Manipulation Language time!

PREV
Chapter 6 : Automating Deployment with SQLCMD

NEXT
Chapter 8 : DML (or Inserts, Updates, and Deletes)