CHAPTER 11

Modifying the Database Structure

In Chapter 2, you saw that you can execute three types of SQL queries. One type is Data Manipulation Language (DML) for querying data. These are the `SELECT`, `INSERT`, `UPDATE`, and `DELETE` queries you've worked with in the previous chapters. Another type is Data Control Language (DCL) queries for controlling access to the database, and you've used the `GRANT` query in the previous chapters to allow the `band` account to access tables and stored procedures in the database. The third type is Data Definition Language (DDL), which allows you to modify the structure of the database.

In Chapter 2, you looked at using SQL Server Management Studio to create the SQL Server database and MySQL Query Browser to create the MySQL database. Both of these graphical clients allow you to manipulate databases, shielding you from the majority of DDL queries. Under the covers, however, DDL queries are used to accomplish the task you specified in the graphical client.

In the intervening chapters, you concentrated on DML queries and built several examples that interacted with the database. In Chapter 10, you looked at a few DDL queries for creating stored procedures: `CREATE PROCEDURE`, `ALTER PROCEDURE`, and `DROP PROCEDURE`.

In this chapter, you'll turn your attention to DDL queries and learn what you can do with them. To work with DDL queries, you need to use a tool that allows you to enter queries and execute them directly against the database. As you've already seen, both SQL Management Studio and MySQL Query Browser allow you to do this. For most of the examples in this chapter, you'll use the graphical clients.

Sometimes, however, you can't use a graphical tool to query the database; in these cases, you need to dive into the murky world of command-line tools. Both SQL Server and MySQL have command-line clients that are installed by default. You'll briefly look at these before you begin examining the various DDL queries.

This chapter is intended as a brief introduction to DDL, not a complete reference work. Entire books have been written about the subject. For examples of more complex DDL, see the scripts provided in this book's code download. You'll see that this chapter doesn't cover a lot of advanced topics (and you'll probably agree that you don't want that much detail at this stage).

This chapter covers the following topics:

- The command-line tools
- DDL for creating databases
- DDL for creating tables
- DDL for adding, modifying, and removing table columns
- DDL for creating and deleting indexes
- DDL for creating and deleting table relationships
- DDL for deleting tables
- DDL for deleting databases

Using Command-Line Tools

Both SQL Server and MySQL install a command-line tool that can connect to both local and remote databases and allow SQL queries to be executed against the database. Although the tools are similar, they have differences. The following sections show each of these in turn.

■ **Note** You have a lot more options for both `SQLCMD` and `mysql.exe` than what you'll see here. You can find more details about `SQLCMD` at `http://msdn.microsoft.com/en-us/library/ms162773.aspx`. For `mysql.exe`, refer to `http://dev.mysql.com/doc/refman/5.0/en/mysql.html`.

Using SQLCMD

`SQLCMD` is installed as part of the SQL Server installation, and if the default installation folder has been accepted, it will be in the `C:\Program Files\Microsoft SQL Server\90\Tools\binn` folder.

To use `SQLCMD`, you must specify the server to connect to and the security credentials you want to use for the connection.

To specify the server, use the `-S` parameter, followed by the server to which you want to connect. If you don't specify a server, `SQLCMD` will assume you want to connect to the default instance of SQL Server on the local machine.

The security credentials you specify depend on whether you're using Windows authentication or SQL authentication. If you want to use Windows authentication, you specify this with the `-E` parameter. If you want to use SQL authentication, you must specify the username and password by using the `-U` and `-P` parameters, respectively. So, to connect to the `(local)\BAND` server using Windows authentication, use the following command line:

```
SQLCMD -E -S (local)\BAND
```

To connect to the same server using SQL authentication and the band account, you need to specify the username and password you want to use, as follows:

```
SQLCMD -U band -P letmein -S (local)\BAND
```

If you're being security conscious, then typing a password in plain text is a big problem. You can force SQLCMD to ask you for the password rather than specifying it on the command line by simply omitting the -P parameter. Then you'll be prompted for the password before the connection is made.

```
SQLCMD -U band -S (local)\BAND
```

Although you can now connect to the server using the correct credentials, you still may not be connected to the correct database. If you don't specify a database, you'll connect to the default database for the user (in most cases, this will be the master database). You can specify the database to use once a connection has been made using the USE SQL query, but it's equally valid to specify this on the command line using the -d parameter followed by the database name. So, to connect to the Players database on the (local)\BAND server using the band account, use the following command:

```
SQLCMD -U band -S (local)\BAND -d Players
```

If the connection to the database is refused, the error message that's returned by SQLCMD is quite helpful. If you specified incorrect login details, you will get a "Login failed for user" message. If you specified an invalid database, you will get a "Cannot open database requested in login" message. You can fix the error noted in the message and then try to connect to the database again.

Try It Out: Querying a SQL Server Database via the Command Line

You'll begin your introduction to DDL using the SQLCMD command-line tool. You'll use this to execute a simple query against the Players database.

1. Open a command prompt, enter the following command, and then press Enter:

   ```
   SQLCMD -U band -S (local)\BAND -d Players
   ```

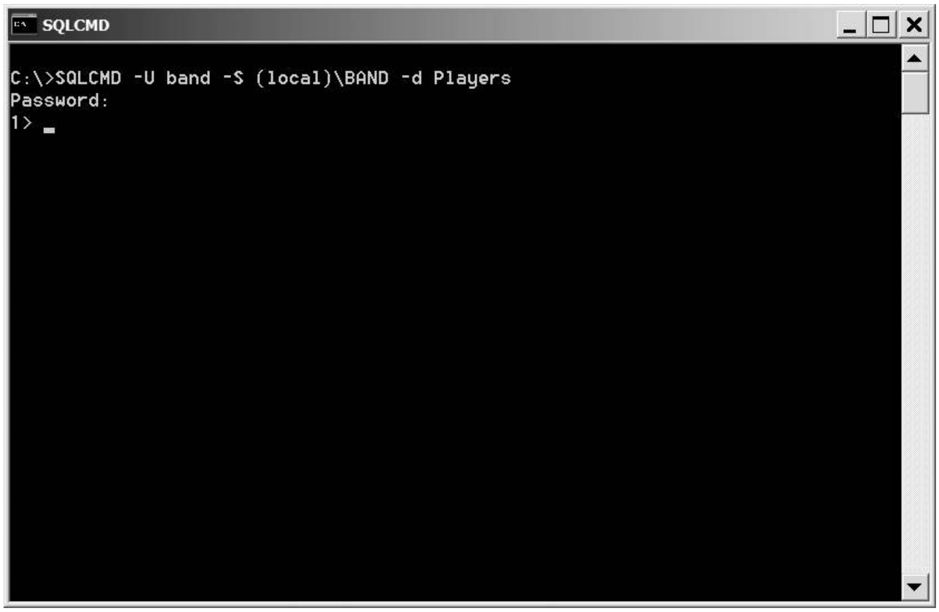
2. At the prompt, enter letmein as the password (your entry won't be echoed to the screen). This will open the SQLCMD command-line tool, as shown in Figure 11-1.

3. Enter the following at the 1> prompt, and then press Enter:

   ```
   SELECT * FROM Manufacturer ORDER BY ManufacturerName
   ```

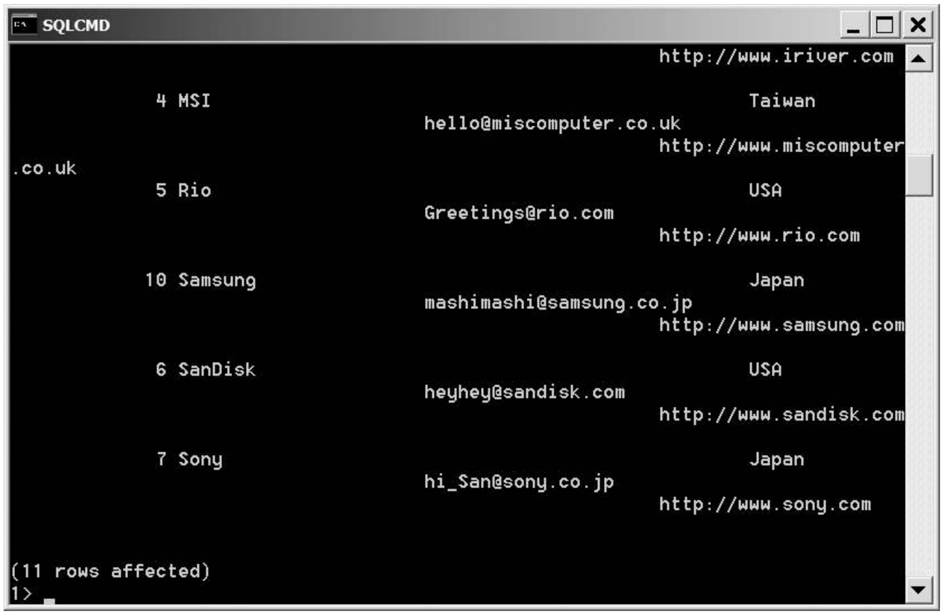
4. At the 2> prompt, enter GO and press Enter. This will execute the command and return the results, as shown in Figure 11-2.

5. Close SQLCMD by entering EXIT or QUIT and pressing Enter.



```
C:\>SQLCMD -U band -S (local)\BAND -d Players
Password:
1> _
```

Figure 11-1. *SQLCMD ready to accept commands*



```
                                                           http://www.iriver.com

             4 MSI                                                 Taiwan
                                     hello@miscomputer.co.uk
                                                           http://www.miscomputer
.co.uk
             5 Rio                                                 USA
                                     Greetings@rio.com
                                                           http://www.rio.com

            10 Samsung                                             Japan
                                     mashimashi@samsung.co.jp
                                                           http://www.samsung.com

             6 SanDisk                                             USA
                                     heyhey@sandisk.com
                                                           http://www.sandisk.com

             7 Sony                                                Japan
                                     hi_San@sony.co.jp
                                                           http://www.sony.com


(11 rows affected)
1> _
```

Figure 11-2. *Query results from SQLCMD*

How It Works

Even though `SQLCMD` is in a rather obscure folder, you can normally launch it from anywhere, because that folder has been added to the command-line execution path.

■**Note** The command-line execution path in Windows allows command-line tools to be executed regardless of the directory that contains the executable and the current directory. The SQL Server installer adds the correct path for `SQLCMD` to the end of the path variable, so you don't normally need to worry about its location.

On the command line, you specify the server, database, and user account you want to use and force `SQLCMD` to ask you for the password you want to use. This prevents anyone from looking over your shoulder and seeing the passwords you're using. This isn't a massive security risk, but you should be doing all you can to ensure that your database password remains secret.

Once you've connected to the database, you execute a simple `SELECT` query against the Manufacturer table and return all the entries in the table, as shown previously in Figure 11-2.

Using `SQLCMD`, you can enter several queries separated by semicolons, or you can spread one query across several different lines (which is quite valuable if you have complex queries). Only when `SQLCMD` sees a `GO` command does it execute the query or queries you've entered.

You close the connection to the database and `SQLCMD` by using either the `QUIT` or `EXIT` command. You can use these commands interchangeably.

Using mysql.exe

The MySQL 5.0 command-line tool is installed in the `bin` folder of the MySQL 5.0 installation. If the defaults have been accepted, it will be in the `C:\Program Files\MySQL\MySQL Server 5.0\bin` folder.

Specifying the server, database, and security credentials for `mysql.exe` is similar to `SQLCMD`.

To specify the server you want to connect to, you use the `-h` parameter, followed by the name of the server. If the server isn't specified, an attempt will be made to connect to MySQL on the local machine.

You specify the security credentials you want to use with the `-u` and `-p` parameters, as with `SQLCMD`, except that with `mysql.exe`, the switches themselves must be lowercase. If you want `mysql.exe` to prompt for the password, specify the `-p` parameter without a value.

You can specify the database you want to connect to by using the `-D` parameter or by simply adding the database name as the last thing on the command line. If you don't specify a database, you'll connect to the server but won't connect to a database; therefore, you must change to the database you want to access with the `USE` query.

So, to connect to the Players database on the local machine using the `band` account, use the following command line:

```
mysql -u band -p Players
```

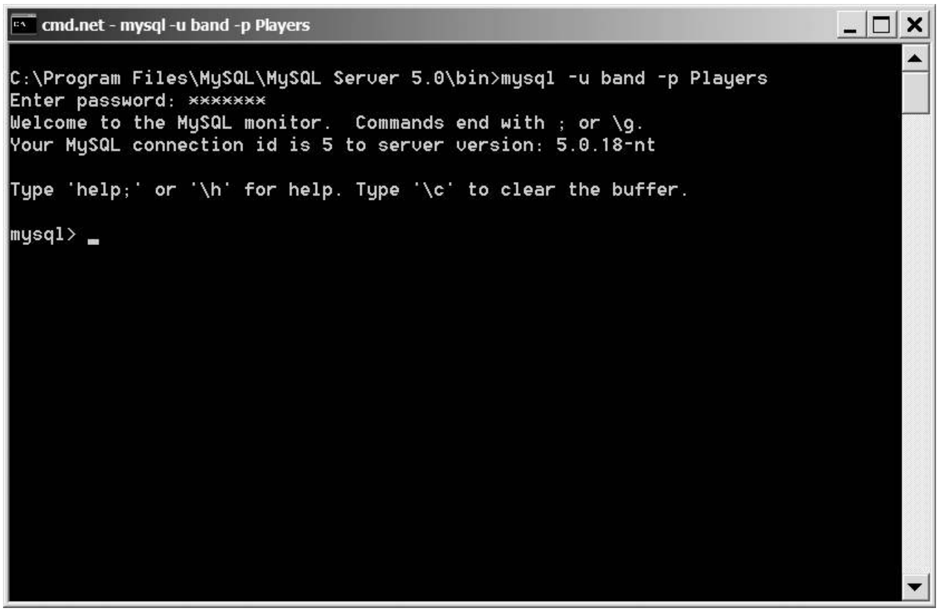
Try It Out: Querying a MySQL Database via the Command Line

In this example, you'll use `mysql.exe` to connect to the Players database and execute a simple query to return all the Manufacturers in the database.

1. Open a command prompt and navigate to the `C:\Program Files\MySQL\MySQL Server 5.0\bin` folder.
2. Enter the following command, and then press Enter:

   ```
   mysql -u band -p Players
   ```

3. At the prompt, enter `letmein` as the password. This will open the `mysql.exe` command-line tool, as shown in Figure 11-3.



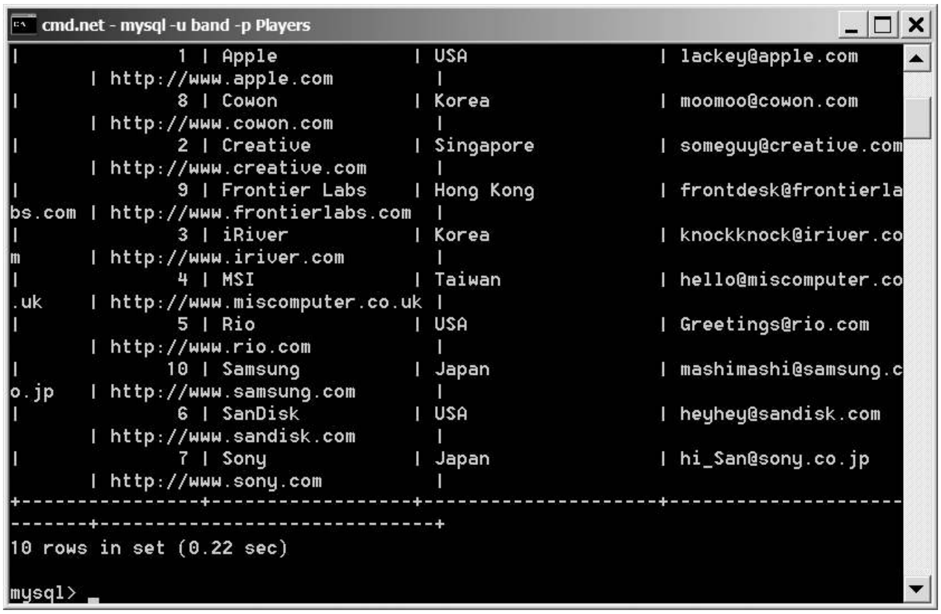
```
C:\Program Files\MySQL\MySQL Server 5.0\bin>mysql -u band -p Players
Enter password: *******
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 5 to server version: 5.0.18-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> _
```

Figure 11-3. *mysql.exe ready to accept commands*

4. Enter the following query, and then press Enter:

   ```
   SELECT * FROM Manufacturer ORDER BY ManufacturerName;
   ```

5. This will execute the query and return the results, as shown in Figure 11-4.
6. Close `mysql.exe` by entering `EXIT` or `QUIT` and then pressing Enter.



```
|              1 | Apple          | USA                  | lackey@apple.com
       | http://www.apple.com       |
|              8 | Cowon          | Korea                | moomoo@cowon.com
       | http://www.cowon.com       |
|              2 | Creative       | Singapore            | someguy@creative.com
       | http://www.creative.com    |
|              9 | Frontier Labs  | Hong Kong            | frontdesk@frontierla
bs.com | http://www.frontierlabs.com |
|              3 | iRiver         | Korea                | knockknock@iriver.co
m      | http://www.iriver.com      |
|              4 | MSI            | Taiwan               | hello@miscomputer.co
.uk    | http://www.miscomputer.co.uk |
|              5 | Rio            | USA                  | Greetings@rio.com
       | http://www.rio.com         |
|             10 | Samsung        | Japan                | mashimashi@samsung.c
o.jp   | http://www.samsung.com     |
|              6 | SanDisk        | USA                  | heyhey@sandisk.com
       | http://www.sandisk.com     |
|              7 | Sony           | Japan                | hi_San@sony.co.jp
       | http://www.sony.com        |
+----------------+----------------+---------------------+--------------------
-------+----------------------------+
10 rows in set (0.22 sec)

mysql> _
```

Figure 11-4. *Query results from mysql.exe*

How It Works

Unlike `SQLCMD`, the path to the `mysql.exe` executable isn't added to the path for you, so you must navigate to the correct folder before you can execute it. You could, of course, use the full path for the executable if you require.

■ **Note** If you're going to use `mysql.exe` often, it's a lot easier to add the path to the executable in the Windows path. You can do this by adding `C:\Program Files\MySQL\MySQL Server 5.0\bin` to the end of the path in Windows in the `autoexec.bat` file or, in Windows 2000 onward, by modifying the `PATH` environment variable.

Once you've entered the password and connected to the database, you enter the queries you want to execute. After they're executed, the results are returned in a tabular format.

Unlike `SQLCMD`, you can enter only one query at a time, but it can again be across multiple lines. Instead of the `GO` command that `SQLCMD` uses, `mysql.exe` uses the semicolon to specify that a query is complete and should be executed.

As with `SQLCMD`, you can also use `QUIT` or `EXIT` to close the connection to the database and exit `mysql.exe`.

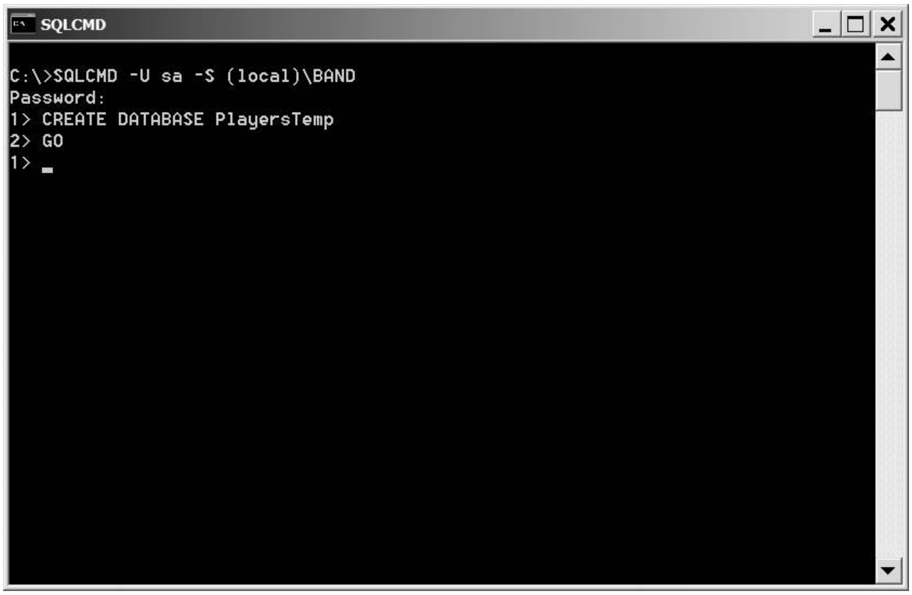
Creating Databases

The first DDL query you need to look at is the query to create a database. In Chapter 2, you briefly learned about `CREATE TABLE`. In Chapter 10, you looked at `CREATE PROCEDURE`. It should come as no surprise that there's a corresponding `CREATE` query for creating databases: `CREATE DATABASE`.

Try It Out: Creating a Database with CREATE DATABASE

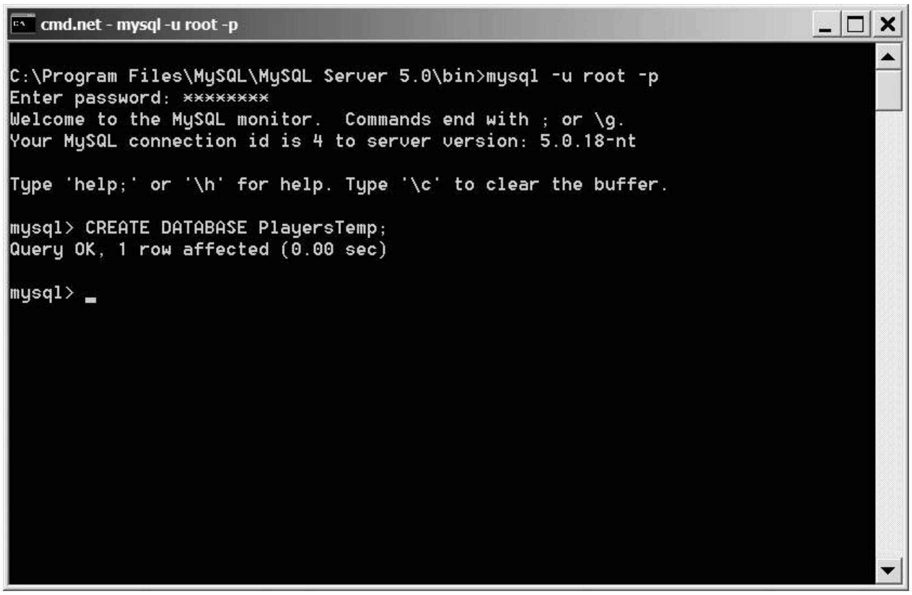
The first DDL query you'll look at is `CREATE DATABASE`. You'll use this to build a database that you'll query in later examples without destroying the database you've been using in the previous chapters. As you've seen, when you use a command-line client, you enter a command and then press Enter to execute it. From now on, the instructions for entering commands won't repeat the "press Enter" part.

1. Open the command-line client for the database you want to use.
   - To connect to SQL Server, open a command prompt and execute the following:
     ```
     SQLCMD -U sa -S (local)\BAND
     ```
   - To connect to MySQL, open a command prompt, navigate to the `C:\Program Files\MySQL\MySQL Server 5.0\bin` folder, and then execute the following command:
     ```
     mysql -u root -p
     ```
2. At the password prompt, enter the correct password, which is bandpass for both SQL Server and MySQL.
3. At the command prompt, enter the query to create the database.
   - For SQL Server, enter the following on two separate lines:
     ```
     CREATE DATABASE PlayersTemp
     GO
     ```
   - For MySQL, enter the following command:
     ```
     CREATE DATABASE PlayersTemp;
     ```
4. The database will be created, and a confirmation message may be returned, depending on whether you're using SQL Server or MySQL.
   - For SQL Server, the database will be created, and no confirmation other than the lack of an error message is given, as shown in Figure 11-5.
   - For MySQL, you get a message saying that the query you've executed is correct, as shown in Figure 11-6.
5. Exit the command-line tool by entering either `EXIT` or `QUIT`.



```
C:\>SQLCMD -U sa -S (local)\BAND
Password:
1> CREATE DATABASE PlayersTemp
2> GO
1> _
```

Figure 11-5. *Database created successfully using SQLCMD*



```
C:\Program Files\MySQL\MySQL Server 5.0\bin>mysql -u root -p
Enter password: ********
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 4 to server version: 5.0.18-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> CREATE DATABASE PlayersTemp;
Query OK, 1 row affected (0.00 sec)

mysql> _
```

Figure 11-6. *Database created successfully using mysql.exe*

6. To verify that the databases have been created correctly, fire up the graphical tool you've been using. You'll see the database has been created.

   - Open SQL Server Management Studio, and you'll immediately see the PlayersTemp database, along with the Players database you've been using, as shown in Figure 11-7.

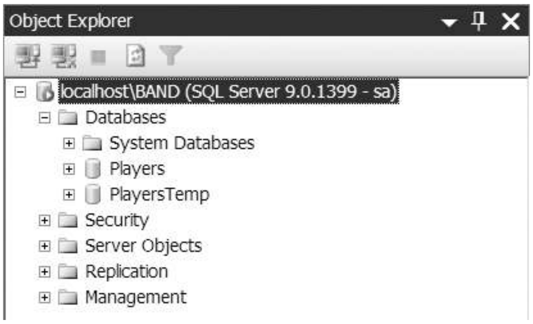


Figure 11-7. *SQL Server Management Studio showing that the database has been created*

   - Open MySQL Query Browser, and you'll immediately see the PlayersTemp database, along with the Players database you've been using, as shown in Figure 11-8.

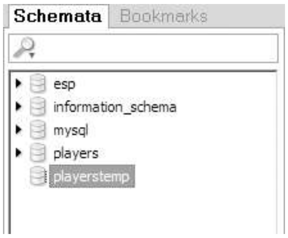


Figure 11-8. *MySQL Query Browser showing that the database has been created*

7. Close both graphical clients.

How It Works

The syntax for the query to create a database is the same whether you're using SQL Server or MySQL. As you'll see shortly, this is one of the few instances where you can use the same query with both SQL Server and MySQL.

Notice that before you execute the query, you're logged in to the server as the administrator account (the `sa` account for SQL Server and the `root` account for MySQL), rather than the `band` account you've been using to execute DML queries. As you saw in Chapter 2, you should always use an account that has only the privileges you need, and the ability to execute DDL queries should be one of the most jealously guarded privileges. You use the administrator account because this is the only account in the system that has permission to execute the DDL queries you need.

To create a database, you use the `CREATE` query, specifying that you want to create a database, and then follow this with the name of the database you want to create, as follows:

```
CREATE DATABASE PlayersTemp
```

So, with this one line of SQL, you're constructing a PlayersTemp database. When you execute this query, the database structure is created before control is returned to the client.

■ **Note** Although this version of `CREATE DATABASE` is the simplest you can get, you can apply a plethora of options to the query to modify how the database is constructed, and these options are different for SQL Server and MySQL. If you're interested, you can find further details for SQL Server at `http://msdn.microsoft.com/en-us/library/ms176061.aspx` and for MySQL at `http://dev.mysql.com/doc/refman/5.0/en/create-database.html`.

Now that you've created the database, you can create the tables that make up the database.

Creating Tables

You briefly saw in Chapter 2 that `CREATE TABLE` is the DDL query for creating tables. For those of you who've been waiting for a fuller discussion, your prayers will now be answered!

The basic structure of the `CREATE TABLE` command is as follows:

```
CREATE TABLE <table-name>
(
  <column1-name column1-type column1-options>,
  <column2-name column2-type column2-options>,
  ...
  <columnN-name columnN-type columnN-options>,
  <table-options>
)
```

You specify the name of the table and then specify each of the columns that make up the table by giving the name of the column, the type of the column, and any other options you need for the column.

In the following examples, you'll create the four tables of the Players database in both SQL Server and MySQL using the graphical client you have available. As you'll see, the queries to create the tables are similar for the two databases, but they're different enough to warrant their own examples.

Try It Out: Creating Tables in SQL Server with CREATE TABLE

In this example, you'll switch from using the command-line to using SQL Server Management Studio. You'll use this graphical client to execute four `CREATE TABLE` queries to create the four tables that make up the PlayersTemp database.

1. Open SQL Server Management Studio and connect to the `(local)\BAND` database using the `sa` account.

2. Expand the Databases node in the Object Explorer and select New Query from the context menu for the PlayersTemp database.

3. In the query text area, enter the following:

```
CREATE TABLE Manufacturer (
  ManufacturerID int IDENTITY NOT NULL PRIMARY KEY,
  ManufacturerName varchar(50) NOT NULL,
  ManufacturerEmail varchar(50)
)
```

4. Click the Execute button. This will execute the query, and if all goes well, you'll receive a "Command(s) completed successfully" confirmation at the bottom of the Query Editor dialog box.

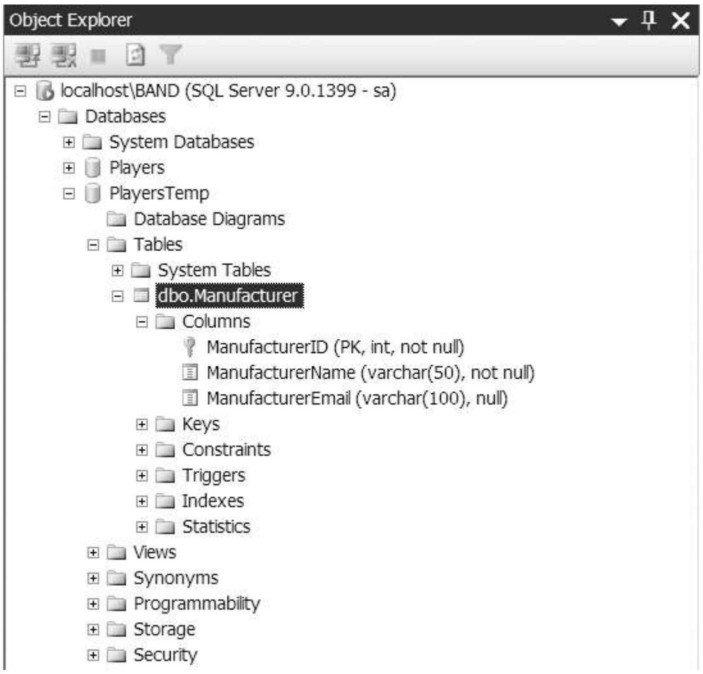
5. To verify that the Manufacturer table has been created, expand the PlayersTemp database in the Object Explorer, and then expand the Tables node. You'll see that the table has indeed been created. Expand the dbo.Manufacturer and Columns nodes, and you'll see that the columns have been created correctly. If the table isn't immediately present, select Refresh from the context menu for the Tables node, and it should appear, as shown in Figure 11-9.

6. Create the Player table by executing the following query:

```
CREATE TABLE Player (
  PlayerID int IDENTITY NOT NULL PRIMARY KEY,
  PlayerName varchar(50) NOT NULL,
  PlayerManufacturerID int NOT NULL
    CONSTRAINT FK_Player_Manufacturer
    REFERENCES Manufacturer(ManufacturerID),
  PlayerStorePrice decimal(10,2) NOT NULL,
  PlayerStorage varchar(50) NOT NULL
)
```

7. Create the Format table by executing the following query:

```
CREATE TABLE Format (
  FormatID int IDENTITY NOT NULL,
  FormatName varchar(50) NOT NULL,
  PRIMARY KEY (FormatID)
)
```



```
localhost\BAND (SQL Server 9.0.1399 - sa)
  Databases
    System Databases
    Players
    PlayersTemp
      Database Diagrams
      Tables
        System Tables
        dbo.Manufacturer
          Columns
            ManufacturerID (PK, int, not null)
            ManufacturerName (varchar(50), not null)
            ManufacturerEmail (varchar(100), null)
          Keys
          Constraints
          Triggers
          Indexes
          Statistics
      Views
      Synonyms
      Programmability
      Storage
      Security
```

Figure 11-9. *After creating a table using SQL Server Management Studio, you can verify that it has been created correctly.*

8. Create the WhatPlaysWhatFormat table by executing the following query:

```
CREATE TABLE WhatPlaysWhatFormat (
  WPWFPlayerID int NOT NULL,
  WPWFFormatID int NOT NULL,
  PRIMARY KEY (WPWFFormatID, WPWFPlayerID)
)
```

9. You can verify that all four tables have been created correctly by refreshing the Tables node in the tree on the left side of the window.

How It Works

The queries you've just used have created all the tables you need.

After naming the table, you define the columns. A column definition contains the name of the column, its data type, and any options you want to apply to that column.

For example, for the Manufacturer table, you're creating the following three columns:

```
ManufacturerID int IDENTITY NOT NULL PRIMARY KEY,
ManufacturerName varchar(50) NOT NULL,
ManufacturerEmail varchar(50)
```

The column name and data type are fairly self-explanatory, but the column options need some explanation.

The IDENTITY option specifies that the column contains an identity value (in other words, one that automatically updates when you insert an entry into the table).

The NOT NULL option specifies that the column doesn't allow null values. By default, a column will accept null values. So, if you want to allow null values, you don't need to specify this option. You can see this if you look at the ManufacturerEmail column:

```
ManufacturerEmail varchar(50)
```

But if you want to be explicit in your definition of whether a column will accept null values, you can also specify this using the NULL option. The following line is functionally equivalent to the previous version:

```
ManufacturerEmail varchar(50) NULL
```

Another column option is PRIMARY KEY, which allows you to specify that the column in question is, not surprisingly, the primary key for the table. If you have a single column making up the primary key, you can specify PRIMARY KEY as a column option, as you do for the Manufacturer table:

```
ManufacturerID int NOT NULL IDENTITY PRIMARY KEY
```

The Player table also uses this method of specifying the primary key. However, you can also specify the primary key as a table option, as you do for the Format table:

```
PRIMARY KEY (FormatID)
```

You can use whichever method of specifying the primary key that you want. However, if the primary key is a composite key and contains more than one column, you must use the alternative syntax, as you do for the WhatPlaysWhatFormat table:

```
PRIMARY KEY (WPWFFormatID, WPWFPlayerID)
```

When specifying the columns for a composite key, you put the column names in a comma-separated list within the brackets of the PRIMARY KEY table option.

The Player table also uses the REFERENCES column option. This allows you to define foreign key constraints at the same time as the table declaration. In this case, you create a relationship called FK_Player_Manufacturer between the PlayerManufacturerID in the Player table and the ManufacturerID in the Manufacturer table:

```
PlayerManufacturerID int NOT NULL
  CONSTRAINT FK_Player_Manufacturer
  REFERENCES Manufacturer(ManufacturerID)
```

You can also add foreign key constraints by specifying a table option, as follows:

```
CONSTRAINT FK_Player_Manufacturer
  FOREIGN KEY (PlayerManufacturerID)
  REFERENCES Manufacturer(ManufacturerID)
```

You can specify both primary and foreign keys on an individual column or as a table option. You're free to choose either option. Personally, I prefer the table option version, as it keeps the keys separate and makes them a little easier to see. So, I would create the Player table as follows:

```
CREATE TABLE Player (
  PlayerID int IDENTITY NOT NULL,
  PlayerName varchar(50) NOT NULL,
  PlayerManufacturerID int NOT NULL,
  PlayerStorePrice decimal(10,0) NOT NULL,
  PlayerStorage varchar(50) NOT NULL,
  PRIMARY KEY (PlayerID),
  CONSTRAINT FK_Player_Manufacturer
    FOREIGN KEY (PlayerManufacturerID)
    REFERENCES Manufacturer(ManufacturerID)
)
```

This version has a few more lines of SQL, but the keys are a lot easier to spot and aren't hidden away among the column definitions.

Although SQL Server gives you the option when defining foreign keys, you'll soon see that MySQL isn't as flexible.

■ **Note** For more information about the options you can supply for columns when using SQL Server, see `http://msdn.microsoft.com/en-us/library/ms174979.aspx`.

Try It Out: Creating Tables in MySQL with CREATE TABLE

In this example, you'll see that the SQL queries to create tables in MySQL are similar to those that you use when creating the tables in SQL Server, but there are differences.

1. Open MySQL Query Browser and connect to the local database using the `root` account.
2. Switch to the PlayersTemp database by double-clicking it in the Schemata pane.
3. Enter the following in the query area:

   ```
   CREATE TABLE Manufacturer (
     ManufacturerID int AUTO_INCREMENT NOT NULL PRIMARY KEY,
     ManufacturerName varchar(50) NOT NULL,
     ManufacturerEmail varchar(50)
   )
   ```

4. Click the Execute icon on the toolbar to execute the query. You'll see a success message, as shown in Figure 11-10.

Query returned no resultset.

Figure 11-10. *MySQL Query Browser provides a little more feedback.*

5. Expand the PlayersTemp database in the Schemata pane. You'll see that the table has been added, as shown in Figure 11-11. If the new table isn't immediately available, refresh the display by selecting Refresh from the PlayersTemp context menu.

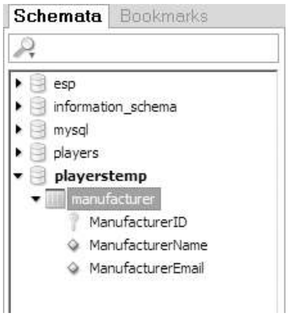


Figure 11-11. *In MySQL Query Browser, you can verify that a table has been created correctly.*

6. Create the Player table by executing the following query:

```
CREATE TABLE Player (
  PlayerID int AUTO_INCREMENT NOT NULL PRIMARY KEY,
  PlayerName varchar(50) NOT NULL,
  PlayerManufacturerID int NOT NULL,
  PlayerStorePrice decimal(10,2) NOT NULL,
  PlayerStorage varchar(50) NOT NULL,
  CONSTRAINT FK_Player_Manufacturer
    FOREIGN KEY (PlayerManufacturerID)
    REFERENCES Manufacturer(ManufacturerID)
)
```

7. Create the Format table by executing the following query:

```
CREATE TABLE Format (
  FormatID int AUTO_INCREMENT NOT NULL,
  FormatName varchar(50) NOT NULL,
  PRIMARY KEY (FormatID)
)
```

8. Create the WhatPlaysWhatFormat table by executing the following query:

```
CREATE TABLE WhatPlaysWhatFormat (
  WPWFPlayerID int NOT NULL,
  WPWFFormatID int NOT NULL,
  PRIMARY KEY (WPWFFormatID, WPWFPlayerID)
)
```

9. You can verify that all four tables have been created correctly by refreshing the Schemata display.

How It Works

As you can see, the queries that you use to create the tables within the database are quite similar to the corresponding queries in SQL Server.

One of the things you need to watch out for is the different names that the different databases use for their data types. While most of the data types will be the same, sometimes the data types have different names or need to be specified slightly differently. See Appendix B for a comparison of the SQL Server and MySQL data types.

The options you can specify in MySQL and SQL Server are similar. The `NOT NULL`, `NULL`, and `PRIMARY KEY` notation for columns operate in the same way for MySQL as they do for SQL Server, and you can also specify primary keys and composite keys using the same terminology. The one column option that's different between SQL Server and MySQL is the `IDENTITY` option in SQL Server. In MySQL, you must specify this type of column using the `AUTO_INCREMENT` option.

The one major difference is the way that foreign keys are defined. Whereas SQL Server allows you to define the foreign key on the column, with MySQL you must specify the foreign keys as a table option. However, the syntax is the same, as in this example:

```
CONSTRAINT FK_Player_Manufacturer
  FOREIGN KEY (PlayerManufacturerID)
  REFERENCES Manufacturer(ManufacturerID)
```

This plays nicely into my preference for creating both primary and foreign keys as table options, rather than intermingled with the column definition:

```
CREATE TABLE Player (
  PlayerID int AUTO_INCREMENT NOT NULL,
  PlayerName varchar(50) NOT NULL,
  PlayerManufacturerID int NOT NULL,
  PlayerStorePrice decimal(10,0) NOT NULL,
  PlayerStorage varchar(50) NOT NULL,
  PRIMARY KEY (PlayerID),
  CONSTRAINT FK_Player_Manufacturer
    FOREIGN KEY (PlayerManufacturerID)
    REFERENCES Manufacturer(ManufacturerID)
)
```

■ **Note** For more information about the options you can supply for columns when you're using MySQL, see `http://dev.mysql.com/doc/refman/5.0/en/create-table.html`.

Adding, Modifying, and Removing Columns

You've now seen how to create tables in both SQL Server and MySQL. This is fine as long as you've created the table correctly in the first place and as long as the requirements for the data that the table will hold don't actually change. If they do change, then you'll need some method of modifying the table.

The basic query to modify a table in the database is `ALTER TABLE`. Depending on what you actually want to do, the other details you must supply will change.

If you want to add a column to a table, you must use the `ADD` syntax of the query, specifying the new column you want to add, like so:

```
ALTER TABLE <table-name> ADD <column-name column-type column-options>
```

You specify the new column in the same way as you do when creating the table. All the options you have available when creating the table are also available here.

To delete a column, you use the `DROP COLUMN` syntax and specify the column you want to delete, like so:

```
ALTER TABLE <table-name> DROP COLUMN <column-name>
```

Sometimes, you'll also need to change the definition of a column. You accomplish this slightly differently in SQL Server and MySQL.

For SQL Server, you use the `ALTER COLUMN` syntax and specify the old column name and the new definition for the column, like so:

```
ALTER TABLE <table-name> ALTER COLUMN <column-name> <column-type column-options>
```

MySQL uses the `CHANGE COLUMN` syntax for this. You must specify the old name of the column, as well as the complete definition, including the column name, for the modified column, like so:

```
ALTER TABLE <table-name> CHANGE COLUMN <column-name>
  <column-name column-type column-options>
```

Be careful when modifying columns, because it's easy to lose data if you don't think things through fully—whether this is from a data-type conversion that was unintended or because a column's length has been reduced. Neither SQL Server nor MySQL will warn that this is about to occur; both will just assume you know what you're doing.

■ **Note** The differences between the SQL Server and MySQL syntax for modifying columns is because this functionality isn't defined in the SQL specification. Most databases allow this functionality, but their implementations are all slightly different.

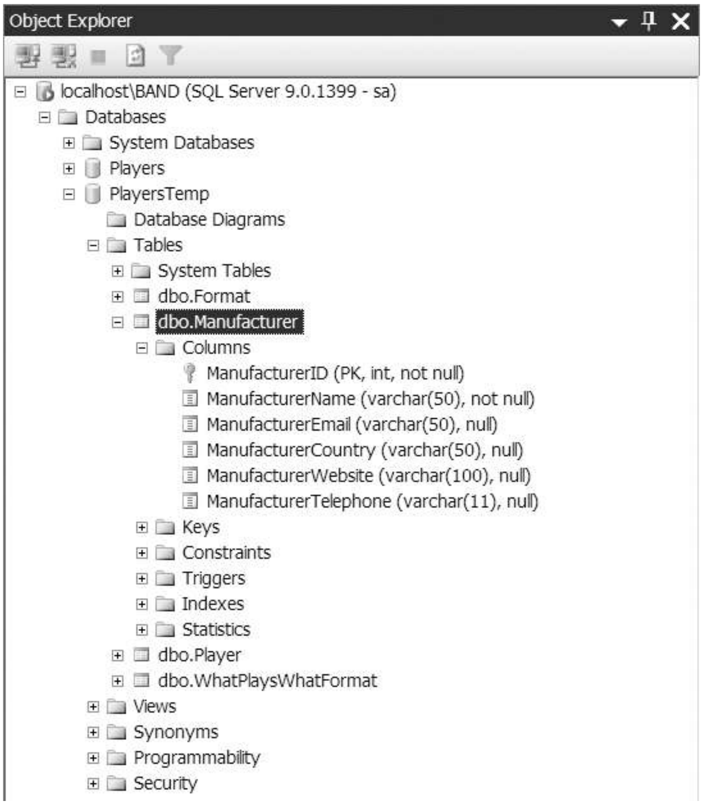
Try It Out: Changing a Table Definition with ALTER TABLE

In this example, you'll modify the Manufacturer table by adding two columns. You'll then modify one of the columns to increase the amount of information that can be stored within the column.

1. Open either SQL Server Management Studio or MySQL Query Browser and connect to the correct database.
   - For SQL Server Management Studio, connect to the `(local)\BAND` database using the `sa` account.
   - For MySQL Query Browser, connect to the `localhost` database using the `root` account.
2. Make sure you're working in the PlayersTemp database.
   - Expand the Databases node in the Object Explorer and select New Query from the context menu for the PlayersTemp database.
   - For MySQL Query Browser, select the PlayersTemp database by double-clicking its name in the Schemata pane.
3. Create two new columns on the Manufacturer table by executing three queries. The queries used by both SQL Server and MySQL are the same in this case, and they should be executed one at a time:

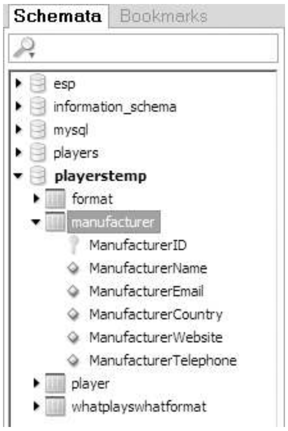
```
ALTER TABLE Manufacturer ADD ManufacturerCountry varchar(50)
ALTER TABLE Manufacturer ADD ManufacturerWebsite varchar(100)
ALTER TABLE Manufacturer ADD ManufacturerTelephone varchar(11)
```

4. Verify that the new columns have been added to the table.
   - For SQL Server Management Studio, expand the PlayersTemp node. Then expand the Tables node and select the Manufacturer table. Expand the Columns node, and you'll see that the new columns have been added and the details match those that you've given, as shown in Figure 11-12.
   - For MySQL Query Browser, expand the PlayersTemp database, and then expand the Manufacturer table. You'll see the columns for the table, including the new ones you've added, as shown in Figure 11-13.



```
Object Explorer
localhost\BAND (SQL Server 9.0.1399 - sa)
  Databases
    System Databases
    Players
    PlayersTemp
      Database Diagrams
      Tables
        System Tables
        dbo.Format
        dbo.Manufacturer
          Columns
            ManufacturerID (PK, int, not null)
            ManufacturerName (varchar(50), not null)
            ManufacturerEmail (varchar(50), null)
            ManufacturerCountry (varchar(50), null)
            ManufacturerWebsite (varchar(100), null)
            ManufacturerTelephone (varchar(11), null)
          Keys
          Constraints
          Triggers
          Indexes
          Statistics
        dbo.Player
        dbo.WhatPlaysWhatFormat
      Views
      Synonyms
      Programmability
      Security
```

Figure 11-12. *Verifying that the new columns have been added to the table in SQL Server Management Studio*



```
Schemata  Bookmarks
esp
information_schema
mysql
players
playerstemp
  format
  manufacturer
    ManufacturerID
    ManufacturerName
    ManufacturerEmail
    ManufacturerCountry
    ManufacturerWebsite
    ManufacturerTelephone
  player
  whatplayswhatformat
```

Figure 11-13. *Verifying that the new columns have been added to the table in MySQL Query Browser*

5. Modify the ManufacturerEmail column on the Manufacturer table by executing the appropriate query.

   - For SQL Server Management Studio, execute the following query:

     ```
     ALTER TABLE Manufacturer ALTER COLUMN ManufacturerEmail varchar(100)
     ```

   - For MySQL Query Browser, execute the following:

     ```
     ALTER TABLE Manufacturer CHANGE COLUMN ManufacturerEmail
       ManufacturerEmail varchar(100);
     ```

6. Verify that the length of the ManufacturerEmail column has indeed changed from 50 characters to 100 characters.

   - For SQL Server Management Studio, you can see the new column width directly in the tree view.

   - For SQL Query Browser, you'll need to look at the table definition. You can see this by selecting Edit Table from the context menu for the Manufacturer table.

7. Execute the command to delete the ManufacturerTelephone column you've just added. The command is the same for both SQL Server and MySQL.

   ```
   ALTER TABLE Manufacturer DROP COLUMN ManufacturerTelephone
   ```

8. Verify that the column has been deleted and that you now have only two new columns in the Manufacturer table: ManufacturerCountry and ManufacturerWebsite.

How It Works

In the example, you first added three columns to the database using the same queries for both SQL Server and MySQL:

```
ALTER TABLE Manufacturer ADD ManufacturerCountry varchar(50)
ALTER TABLE Manufacturer ADD ManufacturerWebsite varchar(100)
ALTER TABLE Manufacturer ADD ManufacturerTelephone varchar(11)
```

Two of these are the ManufacturerCountry and ManufacturerWebsite columns that are in the original database that you didn't include in the `CREATE TABLE` definition for the Manufacturer table in the previous example. The database wouldn't be the same if you didn't have them, so you've added them in this example. You also added a ManufacturerTelephone column that contains a string of up to 11 characters.

You then altered the ManufacturerEmail column for both databases. Fifty characters may not be enough to hold an e-mail address, so you double the size. Unlike adding columns, the two databases have slightly different query forms for this, but they both accomplish the same thing.

For SQL Server, you use the following `ALTER COLUMN` syntax:

```
ALTER TABLE Manufacturer ALTER COLUMN ManufacturerEmail varchar(100)
```

For MySQL, you use the following CHANGE COLUMN syntax:

```
ALTER TABLE Manufacturer CHANGE COLUMN ManufacturerEmail
  ManufacturerEmail varchar(100);
```

Because you're increasing the size of the column, you don't run the risk of losing any data. If you had reduced the size of the column from 100 characters to 50 characters, anything that was contained in the final 50 characters would be permanently lost, so be careful when changing column sizes. Once you've lost the information, you have no way to get it back.

As you probably suspected, you added that column that isn't in the Players database, ManufacturerTelephone, just so you could see how to delete it from the table. You do that with the following query:

```
ALTER TABLE Manufacturer DROP COLUMN ManufacturerTelephone
```

You simply specify the column you want to delete and execute the query. You don't get a confirmation, so make sure you're deleting the correct column before executing the query!

■**Note** Although you've seen perhaps the three most important uses of the `ALTER TABLE` query, you can do a lot more using `ALTER TABLE`. If you want more information, refer to the documentation for SQL Server at `http://msdn.microsoft.com/en-us/library/ms190273.aspx` or for MySQL at `http://dev.mysql.com/doc/refman/5.0/en/alter-table.html`.

Creating and Deleting Indexes

Once you've created the necessary tables in the database, it's possible to create indexes on the tables. You've looked at how to create indexes for both SQL Server and MySQL using graphical tools in Chapter 2 (and learned about the index options in that chapter as well). You saw that the process was quite different for the different tools. However, even though the graphical tools are completely different, both SQL Server and MySQL support the same syntax for creating and deleting indexes.

Creating Indexes

You add indexes to the database using the following CREATE INDEX query:

```
CREATE INDEX <index-name> ON <table-name> ( <column-name> )
```

When creating an index, the first thing you need to specify is a name for the index. SQL Server defaults to using a name of the form IX_*column*, where *column* is the column that's being indexed. It's best to stick to a consistent naming scheme that is easy to understand if you need to return to the database in the future. The SQL Server naming scheme is as good as any.

You then specify the table you're adding the index to and the column you want to index.

■**Note** Both SQL Server and MySQL automatically add an index to a table when you define a primary key (either as a single column or a composite key) for the table. SQL Server calls this index `PK__tablename`, and MySQL calls it `PRIMARY`. Deleting this index will remove the primary key information from the table, so you probably don't want to delete it.

Try It Out: Creating Indexes with CREATE INDEX

You'll now add indexes to the four tables you created in the previous example. You'll add indexes for three different columns. You'll reuse two of these when we look at relationships shortly. You'll delete the third index in the next example.

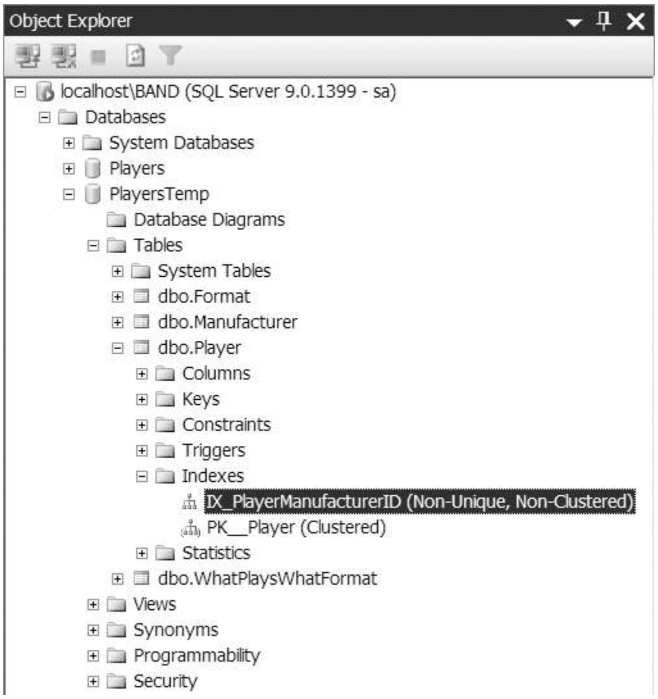
1. Open either SQL Server Management Studio or MySQL Query Browser and connect to the correct database.
   - For SQL Server Management Studio, connect to the `(local)\BAND` database using the `sa` account.
   - For MySQL Query Browser, connect to the `localhost` database using the `root` account.
2. Make sure you're working in the PlayersTemp database.
   - Expand the Databases node in the Object Explorer and select New Query from the context menu for the PlayersTemp database.
   - For MySQL Query Browser, select the PlayersTemp database by double-clicking its name in the Schemata pane.
3. Execute the queries to create the indexes.
   - For SQL Server, you need to execute the following three queries. You'll receive a "Command(s) completed successfully" message for each:

     ```
     CREATE INDEX IX_WPWFPlayerID ON WhatPlaysWhatFormat (WPWFPlayerID)
     CREATE INDEX IX_WPWFFormatID ON WhatPlaysWhatFormat (WPWFFormatID)
     CREATE INDEX IX_PlayerManufacturerID ON Player (PlayerManufacturerID)
     ```

   - For MySQL, you need to execute only the first two queries. You'll receive a "Query returned no resultset" message for each query:

     ```
     CREATE INDEX IX_WPWFPlayerID ON WhatPlaysWhatFormat (WPWFPlayerID)
     CREATE INDEX IX_WPWFFormatID ON WhatPlaysWhatFormat (WPWFFormatID)
     ```

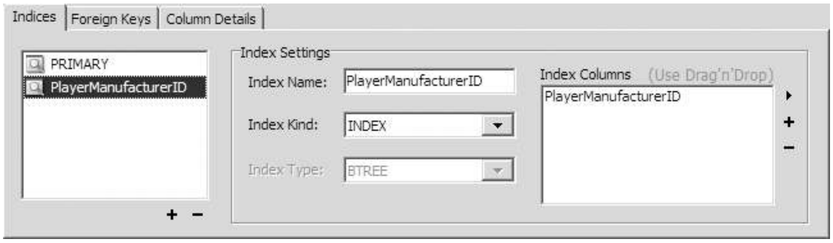
4. You can use the graphical tools to check that the indexes have been created.
   - SQL Server Management Studio has an Indexes node for each table, which shows all the indexes for the table. Figure 11-14 shows the indexes for the Player table.



```
Object Explorer
⊟ localhost\BAND (SQL Server 9.0.1399 - sa)
  ⊟ Databases
    ⊞ System Databases
    ⊞ Players
    ⊟ PlayersTemp
        Database Diagrams
      ⊟ Tables
        ⊞ System Tables
        ⊞ dbo.Format
        ⊞ dbo.Manufacturer
        ⊟ dbo.Player
          ⊞ Columns
          ⊞ Keys
          ⊞ Constraints
          ⊞ Triggers
          ⊟ Indexes
              IX_PlayerManufacturerID (Non-Unique, Non-Clustered)
              PK__Player (Clustered)
          ⊞ Statistics
        ⊞ dbo.WhatPlaysWhatFormat
      ⊞ Views
      ⊞ Synonyms
      ⊞ Programmability
      ⊞ Security
```

Figure 11-14. *Indexes are shown in the tree for SQL Server Management Studio.*

- MySQL Query Browser allows you to see the indexes that have been created by viewing the table definition, as shown in Figure 11-15.



Indices | Foreign Keys | Column Details

PRIMARY
PlayerManufacturerID

Index Settings
Index Name: PlayerManufacturerID
Index Kind: INDEX
Index Type: BTREE
Index Columns (Use Drag'n'Drop)
PlayerManufacturerID

Figure 11-15. *MySQL Query Browser shows the indexes in the Table Editor.*

How It Works

As you can see, the `CREATE INDEX` queries are the same for both SQL Server and MySQL. You simply specify the name of the index, on what table you want the index created, and the columns in the index:

```
CREATE INDEX IX_WPWFPlayerID ON WhatPlaysWhatFormat (WPWFPlayerID)
```

You created three indexes in SQL Server and two indexes in MySQL. Yet MySQL actually has the index that you didn't create already defined, as you can see in Figure 11-15.

If you recall from Chapter 2, MySQL requires an index for any column that is a foreign key, and the column must be the first column in the index. When you added the foreign key in the `CREATE TABLE` query for the Player table, the index for the PlayerManufacturerID column was added automatically.

The other indexes that you added to the WhatPlaysWhatFormat table are required for adding relationships, as you'll do after the discussion of indexes. By creating them now, you've saved the work of having to create them later. However, you need to see how to delete indexes, so one of them has to go.

■**Note** You can find more information about the `CREATE INDEX` command for SQL Server at `http://msdn.microsoft.com/en-us/library/ms188783.aspx` and for MySQL, at `http://dev.mysql.com/doc/refman/5.0/en/create-index.html`.

Deleting Indexes

To delete an index, you use the `DROP INDEX` query. For SQL Server, the syntax is as follows:

```
DROP INDEX <table-name>.<index-name>
```

For MySQL, the syntax is slightly different:

```
DROP INDEX <index-name> ON <table-name>
```

In both versions, you must specify the name of the index and the table for the index. This will drop the index from the specified table without any warning, and you can't recover an index that has been deleted.

Try It Out: Deleting an Index with DROP INDEX

In this example, you'll delete one of the indexes you created in the previous example.

1. Open either SQL Server Management Studio or MySQL Query Browser and connect to the correct database.
   - For SQL Server Management Studio, connect to the `(local)\BAND` database using the `sa` account.
   - For MySQL Query Browser, connect to the `localhost` database using the `root` account.
2. Make sure you're working in the PlayersTemp database.
   - Expand the Databases node in the Object Explorer and select New Query from the context menu for the PlayersTemp database.
   - For MySQL Query Browser, select the PlayersTemp database by double-clicking its name in the Schemata pane.

3. Execute the query to drop the index.

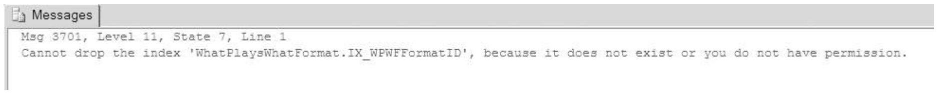
   - For SQL Server, execute this:

     ```
     DROP INDEX WhatPlaysWhatFormat.IX_WPWFFormatID
     ```

   - For MySQL, execute this:

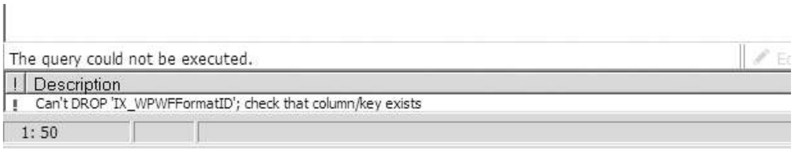
     ```
     DROP INDEX IX_WPWFFormatID ON WhatPlaysWhatFormat
     ```

4. You can verify that the index has been deleted by looking at the indexes for the table, as you did in the previous example. As an alternative, you can also execute the `DROP INDEX` query again. For SQL Server, you'll receive an error message explaining that the index doesn't exist, as shown in Figure 11-16. Figure 11-17 shows the error message for MySQL.



```
Msg 3701, Level 11, State 7, Line 1
Cannot drop the index 'WhatPlaysWhatFormat.IX_WPWFFormatID', because it does not exist or you do not have permission.
```

Figure 11-16. *Unknown index error message in SQL Server*



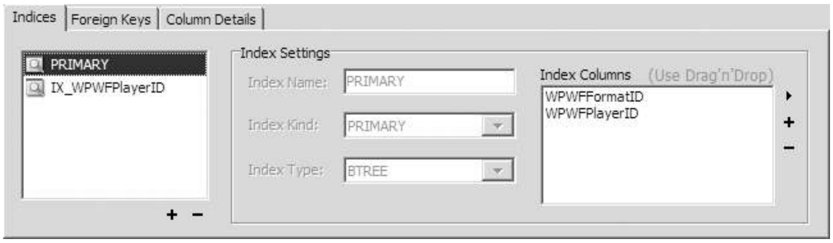
```
The query could not be executed.
! Description
! Can't DROP 'IX_WPWFFormatID'; check that column/key exists
1: 50
```

Figure 11-17. *Unknown index error message in MySQL*

How It Works

As you can see, deleting indexes is easy. Notice that you don't get any warning that you're about to delete an index. However, in this case, the lack of warning isn't really a problem, because you can simply re-create the index without any loss of data. When you look at deleting tables and databases later in this chapter, you'll see that the lack of warning can be a problem, though.

You may have noticed that you've deleted an index that I said was necessary, in MySQL, for the relationships that you are going to add. The key is the fact that the foreign key must be the first column in an index. If you look at Figure 11-18, you'll see that the WPWFFormatID column was already the first key in the `PRIMARY` index.



PRIMARY
IX_WPWFPlayerID

Index Name: PRIMARY
Index Kind: PRIMARY
Index Type: BTREE

WPWFFormatID
WPWFPlayerID

Figure 11-18. *The WPWFFormatID column was already in a suitable index.*

Recall that you created the WhatPlaysWhatFormat table with the following primary key definition:

```
PRIMARY KEY (WPWFFormatID, WPWFPlayerID)
```

A primary key always has an index created for it, and as you've specified the WPWFFormatID column first in the primary key, it becomes the first column in the index. Therefore, you don't need a separate index for it, so it's okay to delete the unnecessary index.

■ **Note** You can find more information about the `DROP INDEX` command for SQL Server at `http://msdn.microsoft.com/en-us/library/ms176118.aspx` and for MySQL at `http://dev.mysql.com/doc/refman/5.0/en/drop-index.html`.

Creating and Deleting Relationships

In Chapter 2, you looked at the relationships between tables and saw how these relationships give databases their real power. After all, they wouldn't be called relational databases unless there was a relation in there somewhere.

Relationships aren't the be-all and end-all of databases, however. You can run a perfectly acceptable database solution without implementing any relationships in the database, and instead relying on the SQL queries you write. While this is perfectly acceptable and a lot of databases have no relationships explicitly defined, you should always use relationships if the database you're using implements them.

Creating Relationships

As you saw in Chapter 2, a relationship will exist between two columns. For the Player database, for example, you have a relationship between a Player and a Manufacturer—the PlayerManufacturer column in the Player table and the ManufacturerID column in the Manufacturer table. You added this relationship when you created the Player table by adding a `FOREIGN KEY` table option to the `CREATE TABLE` query.

It is also possible to add relationships to existing tables. To do this, you need a SQL query that can model this relationship. In this case, you can use the `ADD CONSTRAINT` syntax of the `ALTER TABLE` query.

■ **Note** The `ADD CONSTRAINT` version of the `ALTER TABLE` query allows you to do more than simply add relationships to the table. You can add primary keys to tables using this method, and you can also specify indexes and keys that contain multiple columns using this command. For more information, refer to the `ALTER TABLE` syntax at `http://msdn.microsoft.com/en-us/library/ms190273.aspx` for SQL Server and at `http://dev.mysql.com/doc/refman/5.0/en/alter-table.html` for MySQL.

The syntax of the ADD CONSTRAINT version of the ALTER TABLE query is as follows:

```
ALTER TABLE <table-name>
  ADD CONSTRAINT <relationship-name>
  FOREIGN KEY ( <column-name> )
  REFERENCES <table-name> ( <column-name> )
```

This is a little more involved than the other DDL queries you've seen so far in this chapter, but it isn't that complex. If you refer to the CREATE TABLE queries earlier in this chapter, you'll see that it's very similar to the way that you added relationships when creating the table. Let's look at the query line by line.

First, you need to specify the table to which the relationship is being applied. You always apply a constraint to the foreign key side of the relationship, so for the Player to Manufacturer relationship, you add the constraint to the Player table, because you have multiple Players for only one Manufacturer.

You must then specify a name for the relationship. If you don't have a name, you can't ever refer to it, and in the database everything must have a name. SQL Server defaults to using a name of the form FK_*table1*_*table2*, where *table1* is the table containing the foreign key and *table2* is the table containing the primary key, but you're free to use whatever name you want (limited by the SQL naming conventions, of course). Again, for maintenance reasons, you should stick to a consistent naming scheme, and the SQL Server naming scheme is acceptable.

You then tell the database that you're creating a foreign key (using the fairly obvious FOREIGN KEY syntax) and specify, in brackets, which column you want as the foreign key.

Finally, you must let the database know with which table and column you're creating the relationship. You do this after the REFERENCES clause, and you specify the table name followed by the column name in brackets.

That's it. You now know how to add a relationship between two tables. Theory is one thing, but now you'll actually do it.

Try It Out: Creating Relationships with ALTER TABLE

In this example, you'll add the relationships that exist in the database using the ADD CONSTRAINT version of the ALTER TABLE query.

1. Open either SQL Server Management Studio or MySQL Query Browser and connect to the correct database.
   - For SQL Server Management Studio, connect to the (local)\BAND database using the sa account.
   - For MySQL Query Browser, connect to the localhost database using the root account.
2. Make sure you're working in the PlayersTemp database.
   - Expand the Databases node in the Object Explorer and select New Query from the context menu for the PlayersTemp database.
   - For MySQL Query Browser, select the PlayersTemp database by double-clicking its name in the Schemata pane.

3. Add the relationship between the WhatPlaysWhatFormat and Player tables by executing the following query:

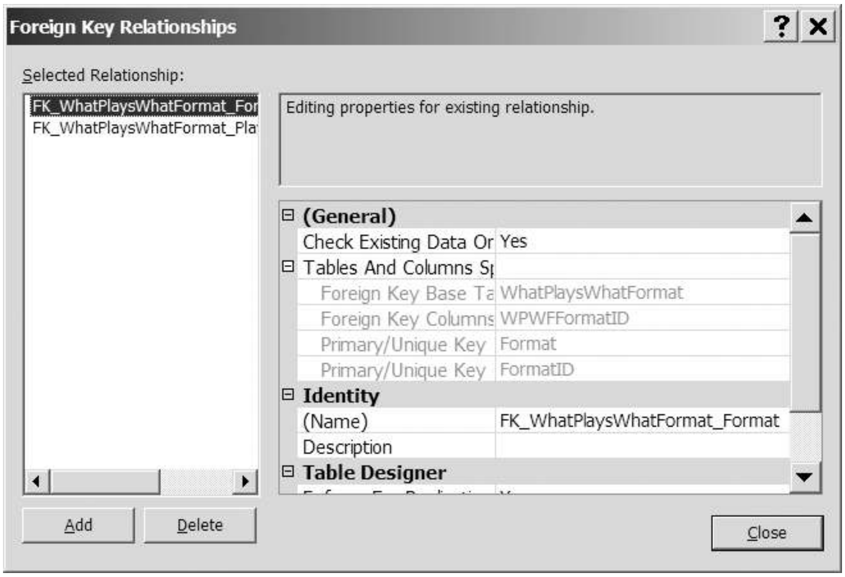
```
ALTER TABLE WhatPlaysWhatFormat
ADD CONSTRAINT FK_WhatPlaysWhatFormat_Player
FOREIGN KEY (WPWFPlayerID)
REFERENCES Player (PlayerID)
```

4. Add the final relationship, between the WhatPlaysWhatFormat and Format tables, by executing the following query:

```
ALTER TABLE WhatPlaysWhatFormat
ADD CONSTRAINT FK_WhatPlaysWhatFormat_Format
FOREIGN KEY (WPWFFormatID)
REFERENCES Format (FormatID)
```

5. You can verify that the relationships have been created correctly by using the graphical tools.

   - To view the relationships for SQL Server, you can either create a diagram containing the related tables (as you learned in Chapter 2) or use SQL Server Management Studio to view the relationships. If you view the table definition, you can select Relationships from the context menu to view the relationships for the table, as shown in Figure 11-19.



Foreign Key Relationships

Selected Relationship:

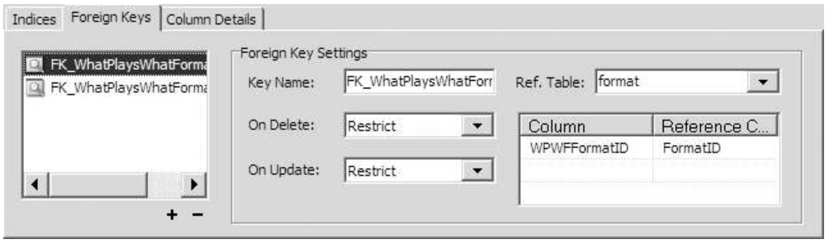
FK_WhatPlaysWhatFormat_For
FK_WhatPlaysWhatFormat_Pla

Editing properties for existing relationship.

⊟ (General)	
Check Existing Data Or	Yes
⊟ Tables And Columns Sp	
Foreign Key Base Ta	WhatPlaysWhatFormat
Foreign Key Columns	WPWFFormatID
Primary/Unique Key	Format
Primary/Unique Key	FormatID
⊟ Identity	
(Name)	FK_WhatPlaysWhatFormat_Format
Description	
⊟ Table Designer	

Figure 11-19. *Relationships can be viewed in SQL Server Management Studio.*

   - You can view the relationships for MySQL using MySQL Query Browser. As shown in Figure 11-20, the Foreign Keys tab of the Edit Table dialog box shows all of the relationships where the table is the foreign key of the relationship.



FK_WhatPlaysWhatForma
FK_WhatPlaysWhatForma

Foreign Key Settings

Key Name: FK_WhatPlaysWhatForr
Ref. Table: format
On Delete: Restrict
On Update: Restrict

Column	Reference C...
WPWFFormatID	FormatID

Figure 11-20. *MySQL Query Browser shows only foreign key relationships.*

How It Works

In this example, you've added the two relationships you require to the database using two separate ALTER TABLE queries.

The first query adds the relationship between Players and WhatPlaysWhatFormat:

```
ALTER TABLE WhatPlaysWhatFormat
ADD CONSTRAINT FK_WhatPlaysWhatFormat_Player
FOREIGN KEY (WPWFPlayerID)
REFERENCES Player (PlayerID)
```

You're creating a relationship between the WPWFPlayerID column in the WhatPlaysWhatFormat table and the PlayerID column in the Player table.

The second query follows a similar pattern. You're creating the relationship between the WPWFFormatID column in the WhatPlaysWhatFormat table and the FormatID column in the Format table:

```
ALTER TABLE WhatPlaysWhatFormat
ADD CONSTRAINT FK_WhatPlaysWhatFormat_Format
FOREIGN KEY (WPWFFormatID)
REFERENCES Format (FormatID)
```

These two relationships that the WhatPlaysWhatFormat table has with the Player and Format tables define the many-to-many relationship between the Player and Format tables. The relationships are both created on the WhatPlaysWhatFormat table because a Player can support several Formats and a Format will be supported by several different Players.

Finally, you viewed the relationships defined on the tables. SQL Server Management Studio allows you to view relationships in two ways. When you learned about database diagrams in Chapter 2, you saw that when a diagram contains two tables that are related, the relationship is shown. The relationships are also visible in the Relationships dialog box, as in Figure 11-19. You can also see that a column is a foreign key by looking at the columns in the Object Explorer. As shown in Figure 11-21, columns that are foreign keys are shown with a gray key icon.

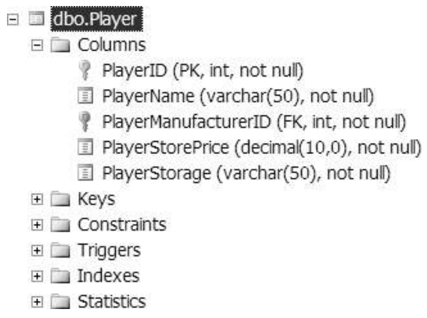

dbo.Player
Columns
PlayerID (PK, int, not null)
PlayerName (varchar(50), not null)
PlayerManufacturerID (FK, int, not null)
PlayerStorePrice (decimal(10,0), not null)
PlayerStorage (varchar(50), not null)
Keys
Constraints
Triggers
Indexes
Statistics

Figure 11-21. *SQL Server Management Studio shows foreign keys in Object Explorer.*

MySQL Query Browser provides fewer tools for viewing the relationships than SQL Server Management Studio offers. Although you can't view the relationships where the table contains the primary key, you can view relationships for a table if the table contains the foreign key column.

Deleting Relationships

Once you've created relationships in the database, you may need to modify those relationships at some point in the future. Unlike with table columns, there's no concept of modifying a relationship in the database. If you need to change a relationship, you must delete the old one and then create the new one.

Deleting a relationship is slightly different depending on whether you're using SQL Server or MySQL, because they refer to relationships as different things and use different queries to delete relationships.

In SQL Server, a relationship is called a *constraint*, and you use the `DROP CONSTRAINT` version of the `ALTER TABLE` query, like so:

```
ALTER TABLE <table-name> DROP CONSTRAINT <relationship-name>
```

MySQL calls a relationship a foreign key and uses the `DROP FOREIGN KEY` version of `ALTER TABLE`, like so:

```
ALTER TABLE <table-name> DROP FOREIGN KEY <relationship-name>
```

You can simply drop a relationship by specifying the table that the relationship was created on and specifying the name of the relationship you want to delete.

■ **Note** If you try to delete a relationship from the wrong table—in other words, from the primary key side of the relationship—an error will be thrown, as the relationship isn't defined on that table. But if you've followed my advice and used a consistent naming scheme that is easy to decipher, you can avoid a lot of these errors.

Try It Out: Deleting Relationships with ALTER TABLE

In this example, you'll start the process of dismantling the database you've built through the previous examples. You'll first delete the relationships in the database.

1. Open either SQL Server Management Studio or MySQL Query Browser and connect to the correct database.
   - For SQL Server Management Studio, connect to the `(local)\BAND` database using the `sa` account.
   - For MySQL Query Browser, connect to the `localhost` database using the `root` account.
2. Make sure you're working in the PlayersTemp database.
   - Expand the Databases node in the Object Explorer and select New Query from the context menu for the PlayersTemp database.
   - For MySQL Query Browser, select the PlayersTemp database by double-clicking its name in the Schemata pane.
3. Drop the relationship between the Player and Manufacturer tables.
   - For SQL Server, execute the following query. If the relationship is deleted correctly you'll receive a "Command(s) completed successfully" message.

     ```
     ALTER TABLE Player DROP CONSTRAINT FK_Player_Manufacturer
     ```
   - For MySQL, execute the following query. You'll receive a "Query returned no resultset" message if the relationship is deleted correctly.

     ```
     ALTER TABLE Player DROP FOREIGN KEY FK_Player_Manufacturer
     ```

How It Works

As in most cases, destroying something is much easier than creating it. To delete relationships, you simply specify the name of the relationship you want to delete and the table to which the relationship belongs. The only wrinkle is that you need to specify the relationship as a `CONSTRAINT` in SQL Server and a `FOREIGN KEY` in MySQL.

Deleting Tables

As with deleting relationships, deleting tables (and, as you'll see shortly, deleting databases) is a lot easier than creating them.

To delete a table from the database, use the `DROP TABLE` query, like so:

```
DROP TABLE <table-name>
```

Executing this query will delete the table without any warning, so be extremely careful that you're deleting the correct table and that deleting the table is indeed what you want to do.

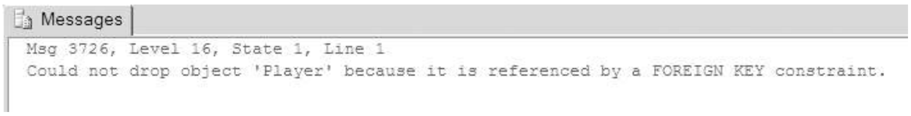
Try It Out: Deleting Database Tables with DROP TABLE

In this example, you'll delete entire tables from the database. You'll delete only three of the four tables that are in the database, as you'll need at least one table for the next example.

1. Open either SQL Server Management Studio or MySQL Query Browser and connect to the correct database.
   - For SQL Server Management Studio, connect to the `(local)\BAND` database using the `sa` account.
   - For MySQL Query Browser, connect to the `localhost` database using the `root` account.
2. Make sure you're working in the PlayersTemp database.
   - Expand the Databases node in the Object Explorer and select New Query from the context menu for the PlayersTemp database.
   - For MySQL Query Browser, select the PlayersTemp database by double-clicking its name in the Schemata pane.
3. Execute the following query to drop the Manufacturer table. You'll get the success message that you've seen in the earlier examples.

   ```
   DROP TABLE Manufacturer
   ```
4. Enter the following query to drop the Player table:

   ```
   DROP TABLE Player
   ```
5. Executing this query will result in an error being thrown.
   - For SQL Server, the error is a `FOREIGN KEY` constraint error, as shown in Figure 11-22.

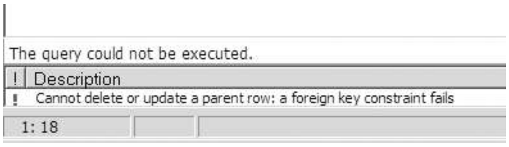


Messages

```
Msg 3726, Level 16, State 1, Line 1
Could not drop object 'Player' because it is referenced by a FOREIGN KEY constraint.
```

Figure 11-22. *SQL Server error if attempting to delete a table in a relationship*

   - For MySQL, the error is a foreign key constraint fails error, as shown in Figure 11-23.



```
The query could not be executed.
! Description
! Cannot delete or update a parent row: a foreign key constraint fails
1: 18
```

Figure 11-23. *MySQL error if attempting to delete a table in a relationship*

6. You can't delete the Player table because it has a relationship with the WhatPlaysWhatFormat table. The WhatPlaysWhatFormat table is the owner of the relationship, so you must delete that table first:

```
DROP TABLE WhatPlaysWhatFormat
```

7. This will delete the relationships owned by the WhatPlaysWhatFormat table, so you can now delete the Player table by executing the same query as you did in step 4:

```
DROP TABLE Player
```

8. To verify that the three tables have been deleted, use the graphical tool to see what tables are left in the database. You should have only one table left.

   - For SQL Server, expand the Object Explorer until you have expanded the Tables node for the PlayersTemp database. You'll see that you still have one table left, Format, as shown in Figure 11-24.

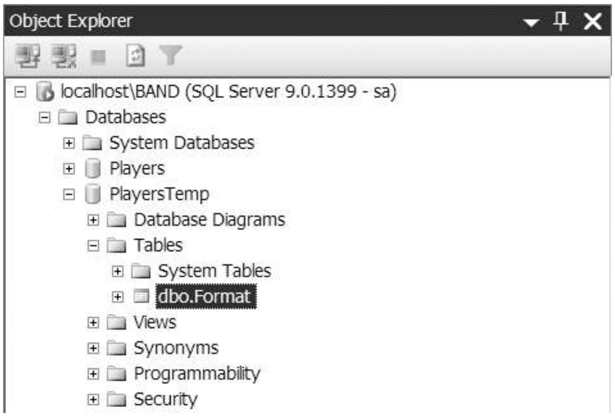


Figure 11-24. *Verifying that the tables have been deleted from the database in SQL Server*

   - For MySQL, refresh the PlayersTemp node in the Schemata pane. You'll see that you have only the Format table remaining, as shown in Figure 11-25.

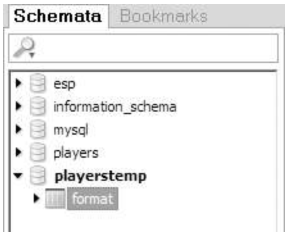


Figure 11-25. *Verifying that the tables have been deleted from the database in MySQL*

How It Works

As you can see, deleting tables is simple and therefore dangerous! Be sure you're deleting the correct table!

The first table you've deleted from the database is the Manufacturer table:

```
DROP TABLE Manufacturer
```

Executing the query successfully deletes the table from the database. When you try to delete the Player table, however, you run into problems.

The Player table provides a primary key for a relationship within the database, so it can't simply be deleted. As you'll recall, you created a relationship between the WhatPlaysWhatFormat and Player table, like so:

```
ALTER TABLE WhatPlaysWhatFormat
ADD CONSTRAINT FK_WhatPlaysWhatFormat_Player
FOREIGN KEY (WPWFPlayerID)
REFERENCES Player (PlayerID)
```

Because the Player table provides the primary key in a relationship defined for the WhatPlaysWhatFormat table, you can't delete the Player table without first deleting the relationship. You could do this using a `DROP CONSTRAINT` or `DROP FOREIGN KEY` command, similar to those you've already looked at, but you use a different solution here.

You can't delete a table if it's providing the primary key in the relationship, but you can delete a table if it's the foreign key in the relationship, and doing so will delete all the foreign key relationships for the table. So, in deleting the WhatPlaysWhatFormat table, you deleted the `FK_WhatPlaysWhatFormat_Player` and `FK_WhatPlaysWhatFormat_Format` relationships. Once these relationships are deleted, you can delete the Player table.

You'll notice that you haven't had any problems deleting tables that have indexes on them. As you'll recall, you specified indexes on the WhatPlaysWhatFormat table, yet the table was deleted without any problems. Unlike relationships, an index deals with only one table, and deleting the table automatically deletes any indexes for the table.

■ **Note** You can find more information about the `DROP TABLE` command for SQL Server at `http://msdn.microsoft.com/en-us/library/ms173790.aspx` and for MySQL at `http://dev.mysql.com/doc/refman/5.0/en/drop-table.html`.

Deleting Databases

If you thought deleting a table was easy and a good way to lose data, then deleting a database is just as easy—and with a lot more scope to delete a lot of data you didn't want deleted. So, you need to be especially careful when you decide to delete a database.

The query to drop a database is as follows:

```
DROP DATABASE <name>
```

This will delete the entire database along with any tables, relationships, and data that still exist in the database. SQL provides no safeguards against deleting entire databases, and this emphasizes why you need to restrict access to the database and, in particular, any administration privileges. If someone has the password for the administrator account, she can delete everything on the server easily. Keep your administrator password secure, and don't give any other account any administrator privileges.

Try It Out: Deleting a Database

To delete the database, you'll now switch back to using the command-line tools to execute a `DROP DATABASE` query. Follow these steps:

1. Open the command-line client for the database you want to use.
   - To connect to SQL Server, open a command prompt and enter the following:

     ```
     SQLCMD -U sa -S (local)\BAND
     ```

   - To connect to MySQL, open a command prompt, navigate to the `C:\Program Files\MySQL\MySQL Server 5.0\bin` folder and execute the following command:

     ```
     mysql -u root -p
     ```

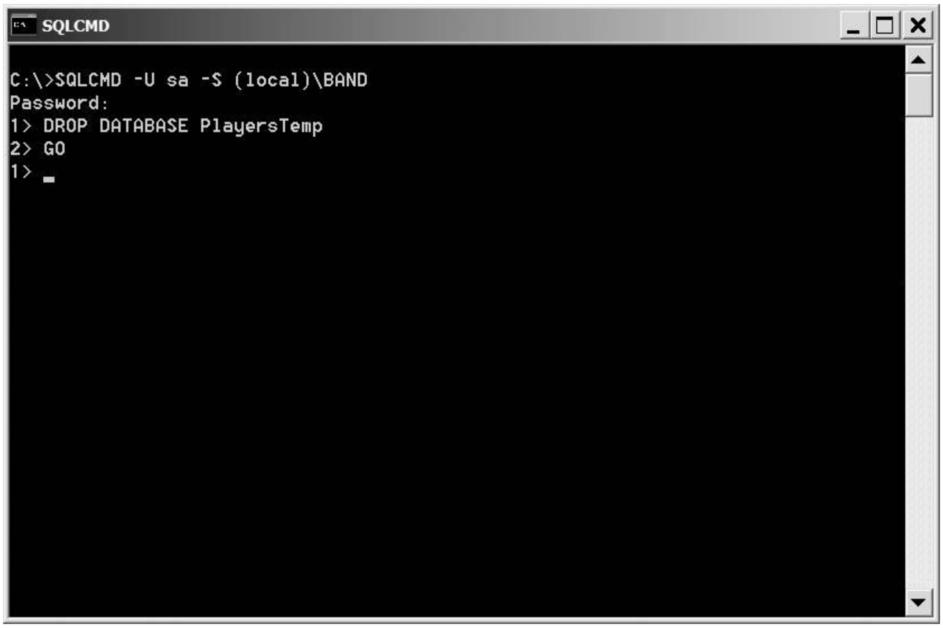
2. At the password prompt, enter the correct password, which is bandpass for both SQL Server and MySQL.
3. At the command prompt, enter the query to delete the database.
   - For SQL Server, enter the following on two separate lines:

     ```
     DROP DATABASE PlayersTemp
     GO
     ```

   - For MySQL, enter the following command:

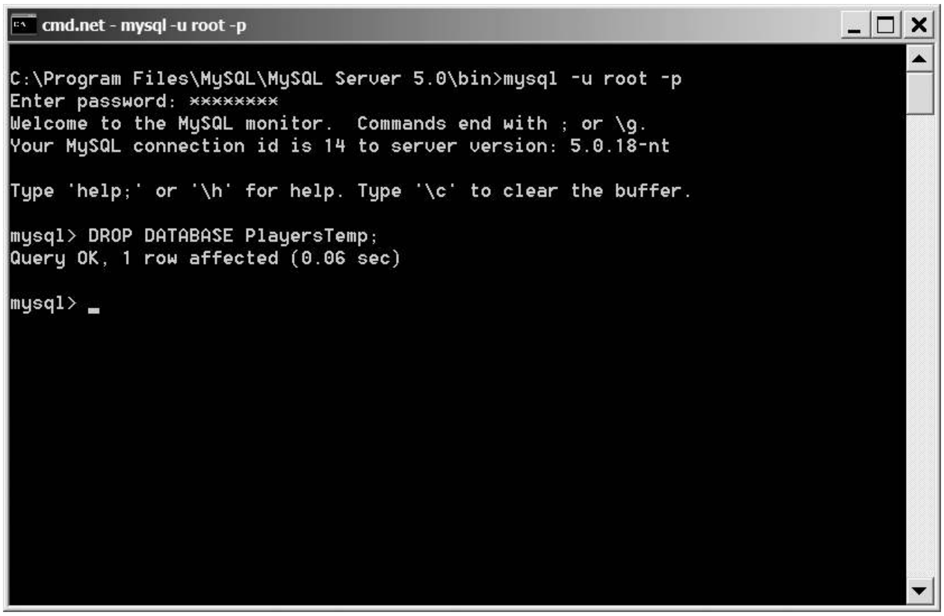
     ```
     DROP DATABASE PlayersTemp;
     ```

4. The database will be deleted. For SQL Server, no confirmation message is given, as shown in Figure 11-26. For MySQL, a brief message is returned, as shown in Figure 11-27.


SQLCMD

```
C:\>SQLCMD -U sa -S (local)\BAND
Password:
1> DROP DATABASE PlayersTemp
2> GO
1> _
```

Figure 11-26. *Database successfully deleted in SQL Server*


cmd.net - mysql -u root -p

```
C:\Program Files\MySQL\MySQL Server 5.0\bin>mysql -u root -p
Enter password: ********
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 14 to server version: 5.0.18-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> DROP DATABASE PlayersTemp;
Query OK, 1 row affected (0.06 sec)

mysql> _
```

Figure 11-27. *Database successfully deleted in MySQL*

How It Works

There isn't an awful lot to say—except be careful! The `DROP DATABASE` query deletes databases with no warning.

■**Note** You can find more information about the `DROP DATABASE` command for SQL Server at `http://msdn.microsoft.com/en-us/library/ms178613.aspx` and for MySQL at `http://dev.mysql.com/doc/refman/5.0/en/drop-database.html`.

Summary

In this chapter, you've taken a whirlwind tour through the DDL "subset" of SQL, yet you've hardly scratched the surface. As I noted at the beginning of the chapter, if you look at the scripts provided in the code download, you'll see that even for this small database, the scripts are complex—certainly more complex than you've seen here.

You've looked at the basics of DDL, and you've seen how to create, modify, and delete databases and tables in both SQL Server and MySQL. You've also seen how you can create indexes on tables and relationships between the different tables in the database.

Remember the following points when using DDL:

- You usually have multiple ways to do the same thing; for example, you looked at two ways to create primary keys on tables. None of the ways to do something is more correct than the others, so choose the one you're comfortable using.
- Be extremely careful when deleting things from the database. It's easy to destroy a table or a database using just a single line of SQL.

You've just about finished with your look at databases, and in the next chapter you'll learn about several topics that will improve how you handle databases. You'll look at concurrency, caching, transactions, and multiple result sets and see that these more advanced topics can improve the Web sites that you're building quite dramatically.