

4

Advanced Database Design

This chapter is all about improving the design of a database in terms of ease of management, efficiency, and limiting the risks of entering invalid data. The chapter begins with a bit of theory and relates it to the practical, in particular the Film Club database. This covers improving efficiency in terms of data storage and minimizing wasted space.

Next, the chapter covers various ways of ensuring that only valid data enters the database. The database can't check the accuracy of what you enter, but it can ensure that what you enter doesn't cause the database to stop working as expected. For example, the Attendance table in the Film Club database relies on its relationship with the Location table. The Attendance table doesn't store the full address details of meetings; it just stores a unique ID that matches a record in the Location table. If, however, no matching record exists in the Location table, then queries return incorrect results. This chapter shows you, among other things, how to enforce the relationship between tables and prevent such a situation from occurring.

You use the things you learn throughout the chapter to update and improve the Film Club database. The chapter finishes off with some tips on things to look out for and things to avoid when designing your database.

Normalization

Chapter 1 discussed how to design the database structure using tables and fields in order to avoid problems such as unnecessary duplication of data and the inability to uniquely identify records. Although not specifically called *normalization* in Chapter 1, that was the concept used. This section explains normalization in more detail and how to use it to create well-structured databases.

Normalization consists of a series of guidelines that help guide you in creating a good database structure. Note that they are guidelines and not rules to follow blindly. Database design is probably as much art as it is science, and your own common sense is important, too.

Normalization guidelines are divided into *normal forms*; think of *form* as the format or the way a database structure is laid out. Quite a lot of normal forms exist, but this chapter examines only the first three, because that's as far as most people need to go in normalizing a database. Taken too far,

Chapter 4

normalization can make database access slower and more complex. The aim of normal forms is to organize the database structure so that it complies with the rules of first normal form, then second normal form, and finally third normal form. It's your choice to take it further and go to fourth normal form, fifth normal form, and so on, but generally speaking, third normal form is enough.

First Normal Form

Chapter 1 walked you through some basic steps for creating a well-organized database structure. In particular, it said that you should do the following:

- ☐ Define the data items required, because they become the columns in a table. Place related data items in a table.
- ☐ Ensure that there are no repeating groups of data.
- ☐ Ensure that there is a primary key.

These rules are those of first normal form, so you've covered first normal form without even knowing it. The Film Club database already complies with first normal form, but to refresh your memory, here is a brief summary of first normal form.

First, you must define the data items. This means looking at the data to be stored, organizing the data into columns, defining what type of data each column contains, and finally putting related columns into their own table. For example, you put all the columns relating to locations of meetings in the Location table, those relating to members in the MemberDetails table, and so on.

This gives a rough idea of table structures. The next step is ensuring that there are no repeating groups of data. If you remember from Chapter 1, when you created the Film Club database, you began by defining a table that held members' details and also details of meetings they had attended. An example of how the table and records might look is shown below:

Name	Date of Birth	Address	Email	Date Joined	Meeting Date	Location	Did Member Attend?
Martin	Feb 27, 1972	1 The Avenue, NY	martin@some.com	Jan 10, 2005	Mar 30, 2005	Lower West Side, NY	Y
Jane	Dec 12, 1967	33 Some Road, Washington	Jane@server.net	Jan 12, 2005	Mar 30, 2005	Lower West Side, NY	N
Kim	May 22, 1980	19 The Road, New Townsville	kim@mail.com	Jan 23, 2005	Mar 30, 2005	Lower West Side, NY	Y

You find, though, that each time you want to add a record of details of another meeting you end up duplicating the members' details:

Name	Date of Birth	Address	Email	Date Joined	Meeting Date	Location	Did Member Attend?
Martin	Feb 27, 1972	1 The Avenue, NY	martin@some.com	Jan 10, 2005	Mar 30, 2005	Lower West Side, NY	Y
Martin	Feb 27, 1972	1 The Avenue, NY	martin@some.com	Jan 10, 2005	April 28, 2005	Lower North Side, NY	Y
Jane	Dec 12, 1967	33 Some Road, Washington	Jane@server.net	Jan 12, 2005	Mar 30, 2005	Lower West Side, NY	N
Jane	Dec 12, 1967	33 Some Road, Washington	Jane@server.net	Jan 12, 2005	April 28, 2005	Upper North Side, NY	Y
Kim	May 22, 1980	19 The Road, New Townsville	kim@mail.com	Jan 23, 2005	Mar 30, 2005	Lower West Side, NY	Y
Kim	May 22, 1980	19 The Road, New Townsville	kim@mail.com	Jan 23, 2005	April 28, 2005	Upper North Side, NY	Y

There are a number of problems with this approach. First, it wastes a lot of storage space. You only need to store the members' details once, so repeating the data is wasteful. What you have is repeating groups, something that the second rule of first normal form tells you to remove. You can remove duplicated data by splitting the data into tables: one for member details, another to hold location details, and a final table detailing meetings that took place. Another advantage of splitting data into tables is that it avoids something called the *deletion anomaly*, where deleting a record results in also deleting the data you want to keep. For example, say you want to delete details of meetings held more than a year ago. If your data isn't split into tables and you delete records of meetings, then you also delete members' details, which you want to keep. By separating member details and meeting data, you can delete one and not both.

After you split data into tables, you need to link the tables by some unique value. The final rule of the first normal form — create a primary key for each table — comes into play. For example, you added a new column, `MemberId`, to the `MemberDetails` table and made it the primary key. You don't have to create a new column, however; you could just use one or more of the existing columns, as long as, when taken together, the data from each column you combine makes for a unique key. However, having an ID column is more efficient in terms of data retrieval.

Second Normal Form

First normal form requires every table to have a primary key. This primary key can consist of one or more columns. The primary key is the unique identifier for that record, and second normal form states that there must be no partial dependences of any of the columns on the primary key. For example, imagine that you decide to store a list of films and people who starred in them. The data being stored is film ID, film name, actor ID, actor name, and date of birth.

You could create a table called `ActorFilms` using those columns:

Chapter 4

Field Name	Data Type	Notes
FilmId	integer	Primary key
FilmName	varchar(100)	
ActorId	integer	Primary key
ActorName	varchar(200)	
DateOfBirth	date	

This table is in first normal form, in that it obeys all the rules of first normal form. In this table, the primary key consists of FilmId and ActorId. Combined they are unique, because the same actor can hardly have starred in the same film twice!

However, the table is not in second normal form because there are partial dependencies of primary keys and columns. FilmName is dependent on FilmId, and there's no real link between a film's name and who starred in it. ActorName and DateOfBirth are dependent on ActorId, but they are not dependent on FilmId, because there's no link between a film ID and an actor's name or their date of birth. To make this table comply with second normal form, you need to separate the columns into three tables. First, create a table to store the film details, somewhat like the Films table in the example database:

Field Name	Data Type	Notes
FilmId	integer	Primary key
FilmName	varchar(100)	

Next, create a table to store details of each actor:

Field Name	Data Type	Notes
ActorId	integer	Primary key
ActorName	varchar(200)	
DateOfBirth	date	

Finally, create a third table storing just FilmIds and ActorIds to keep track of which actors starred in which films:

Field Name	Data Type	Notes
FilmId	integer	Primary key
ActorId	integer	Primary key

Now the tables are in second normal form. Look at each table and you can see that all the columns are tied to the primary key, except that both columns in the third table make up the primary key.

Third Normal Form

Third normal form is the final step this book illustrates. More normal forms exist, but they get more complex and don't necessarily lead to an efficient database. There's a trade-off in database design between minimizing data redundancy and database efficiency. Normal forms aim to reduce the amount of wasted storage and data redundancy. However, they can do so at the cost of database efficiency, in particular, the speed of getting data in and out. Good practice is to ensure that your database tables are in second normal form. Third normal form is a little more optional, and its use depends on the circumstances.

A table is in third normal form when the following conditions are met:

- ☐ It is in second normal form.
- ☐ All nonprimary fields are dependent on the primary key.

The dependency of nonprimary fields is between the data. For example, street name, city, and state are unbreakably bound to the zip code. For example, New Street, New City, Some State, has an assigned and unique zip code. If you mail a letter, in theory supplying the street address and zip code is enough for someone to locate the house. Another example is social security number and a person's name. No direct dependency exists, however, between zip code and a person's name. The dependency between social security number and name and between zip code and address is called a *transitive dependency*.

Take a look at the Film Club database. Can you see any tables not in third normal form?

Check out the MemberDetails table:

Field Name	Data Type
MemberId	integer
FirstName	nvarchar(50)
LastName	nvarchar(50)
DateOfBirth	date
Street	varchar(100)
City	varchar(75)
State	varchar(75)
ZipCode	varchar(12)
Email	varchar(200)
DateOfJoining	date

Chapter 4

The Street, City, and State fields all have a transitive dependency on the ZipCode field. To comply with third normal form, all you need to do is move the Street, City, and State fields into their own table, which you can call the ZipCode table:

Field Name	Data Type
ZipCode	<code>varchar (12)</code>
Street	<code>varchar (100)</code>
City	<code>varchar (75)</code>
State	<code>varchar (75)</code>

Then alter the MemberDetails table and remove the Street, City, and State fields, leaving the ZipCode field as a way of matching the address details in the ZipCode table:

Field Name	Data Type
MemberId	<code>integer</code>
FirstName	<code>nvarchar (50)</code>
LastName	<code>nvarchar (50)</code>
DateOfBirth	<code>date</code>
ZipCode	<code>varchar (12)</code>
Email	<code>varchar (200)</code>
DateOfJoining	<code>date</code>

The advantages of removing transitive dependencies are mainly twofold. First, the amount of data duplication is reduced and therefore your database becomes smaller. Two or more people living on the same street in the same city in the same town have the same zip code. Rather than store all that data more than once, store it just once in the ZipCode table so that only the ZipCode is stored more than once.

The second advantage is data integrity. When duplicated data changes, there's a big risk of updating only some of the data, especially if it's spread out in a number of different places in the database. If address and zip code data were stored in three or four different tables, then any changes in zip codes would need to ripple out to every record in those three or four tables. However, if it's all stored in one table, then you need to change it in only one place.

There is a downside, though: added complexity and reduced efficiency. Before changing the database, the query for a member's name and address would be as follows:

```
SELECT FirstName, LastName, Street, City, State, ZipCode
FROM MemberDetails;
```

After changing the database so that the MemberDetails table is in third normal form, the same query looks like this:

```
SELECT FirstName, LastName, Street, City, State, ZipCode.ZipCode
FROM MemberDetails INNER JOIN ZipCode
ON ZipCode.ZipCode = MemberDetails.ZipCode;
```

In addition to being a little more complex, the code is also less efficient. The database takes longer to retrieve the data. In a small database with a few records, the difference in speed is minimal, but in a larger database with more records, the difference in speed could be quite significant.

Leave the Film Club database's structure intact for now, and make any necessary changes in the section later in this chapter that re-examines the database's structure.

Ensuring Data Validity with Constraints

When creating database tables and fields, you can specify *constraints* that limit what data can go in a field. It might sound a bit odd — why constrain things? Imagine that a database user is working late, entering data into the database. He's all bleary-eyed and kind of keen to get home. When entering primary key data, he accidentally enters the same value twice. Now the unique primary key is no longer unique, causing data corruption in the database and causing queries, which rely on the uniqueness of the primary key, to fail. Sorting out corrupted data can be quite tricky, especially if you're dealing with a huge database and lots of records. Preventing data corruption in the first place is much better — for example, by defining a constraint when creating the table that specifies that a column must contain unique values. If someone tries to add two identical values, the database throws an error and stops them.

Using the DBMS to protect data integrity is not the only way, but it's probably the best. An alternative is having an external program ensure that the data is valid, as most databases are not accessed directly by the user but via some third-party front-end interface. For example, if you buy a program off the shelf to help catalog and organize your CD and record collection, it's quite unlikely that all you'd get is a database! Instead, the program installs a nice, easy-to-use front-end program on the user's computer that accesses a database in the background without the user ever seeing it. This program can control the data and ensure that no data corruption occurs. However, preventing data corruption using the DBMS has its advantages. For example, the check has to be done in only one place: in the database itself. Additionally, the DBMS can run the check more efficiently than can an external program.

This section covers the following constraints:

- ☐ NOT NULL
- ☐ UNIQUE
- ☐ CHECK
- ☐ PRIMARY KEY
- ☐ FOREIGN KEY

Chapter 4

MySQL versions prior to 5.0.2 support the constraints mentioned here but don't enforce them, which somewhat reduces their effectiveness. The code described in the following sections works in MySQL, but when you enter data prohibited by a particular constraint, MySQL won't reject it as other database systems would. Discussion of constraints and invalid data as they relate to MySQL is beyond the scope of this book. Be sure to check the MySQL product documentation for specific information.

NOT NULL Constraint

Chapter 3 briefly covered the `NULL` data type. If you recall, `NULL` is not the same as no data; rather, it represents unknown data. However, the `NULL` data type can cause problems with query results. It makes sorting columns with `NULL` values difficult, because one `NULL` is the same as the next, so all the `NULL`s appear at the start of any sorted results set. You may also want to ensure that a particular column has a value—that it's a compulsory column when inserting a new record. This is where the `NOT NULL` constraint comes in. It ensures that the column must have a value or else the record cannot be inserted.

Whether a column in a table can contain `NULL`s is something that you need to define when you first create the table. SQL allows `NULL`s by default, so only when you don't want `NULL`s do you need to specify such in the table creation SQL. Add the constraint to the column definition, immediately after the data type. For example, the following SQL creates a new table called `MyTable` and adds three columns, two of which, `Column1` and `Column3`, specify not to accept `NULL`s:

```
CREATE TABLE MyTable
(
  Column1 int NOT NULL,
  Column2 varchar(20),
  Column3 varchar(12) NOT NULL
)
```

What if you decide later on, after you create a table, that you want to make one or more of the columns subject to a `NOT NULL` constraint? Answering this question is a little trickier, and this is another good reason for planning database table creation in advance of creating the tables. Some DBMSs, such as Oracle, allow users to modify columns in an existing table with the `ALTER TABLE MODIFY` statement. For example, to add a `NOT NULL` constraint to `Column1` in Oracle and MySQL, you would write a statement similar to the following:

```
ALTER TABLE MyTable
MODIFY Column2 varchar(20) NOT NULL;
```

However, many database systems don't allow this statement, so you must take a different approach: alter the column definition itself and re-create it with the `NOT NULL` constraint:

```
ALTER TABLE MyTable
ALTER COLUMN Column2 varchar(20) NOT NULL;
```

The `ALTER COLUMN` statement changes `Column2`, specifying the same data type but adding the `NOT NULL` constraint.

IBM's DB2 doesn't allow you to alter a column so that it doesn't allow NULLs. You can either delete the table and redefine it, this time making sure the column has a NOT NULL constraint, or add a CHECK constraint. The CHECK constraint is covered later in this chapter, but to add a NOT NULL CHECK constraint in DB2, the syntax is as follows:

```
ALTER TABLE table_name
ADD CONSTRAINT constraint_name
CHECK (column_name IS NOT NULL)
```

So, to add a NOT NULL CHECK constraint for Column2 of MyTable, you would write the following code:

```
ALTER TABLE MyTable
ADD CONSTRAINT Column2_NotNull
CHECK (Column2 IS NOT NULL);
```

With the NOT NULL constraint added to the database systems, test it out by executing the following SQL:

```
INSERT INTO MyTable (Column1, Column3)
VALUES (123, 'ABC');
```

You should get an error message similar to Cannot insert the value NULL into column 'Column2'. The new record isn't added to the table.

Consider this final issue. What if the column in your table that you want to make subject to a NOT NULL constraint already contains NULL records? You can easily deal with that by using a bit of SQL, (like that shown in the following example) to update all those records containing NULLs and set them to whatever default value you think best:

```
UPDATE MyTable
SET Column2 = ''
WHERE Column2 IS NULL;
```

You must do this prior to adding a NOT NULL clause in order to avoid error messages.

If you ran the examples against the Film Club database, delete MyTable from the database, as it's not needed:

```
DROP TABLE MyTable;
```

UNIQUE Constraint

The UNIQUE constraint prevents two records from having identical values in a particular column. In the MemberDetails table, for example, you might want to prevent two or more people from having identical email addresses.

You can apply the UNIQUE constraint in two ways: add it when creating the table, or add it after creating the table.

Chapter 4

Except for IBM's DB2, when creating a table, add the `UNIQUE` constraint to the table definition immediately after the column's type definition:

```
CREATE TABLE MyUniqueTable
(
    Column1 int,
    Column2 varchar(20) UNIQUE,
    Column3 varchar(12) UNIQUE
);
```

The preceding SQL creates a new table called `MyUniqueTable` and adds the `UNIQUE` constraint to `Column2` and `Column3`. If you're using DB2, you can't create a `UNIQUE` constraint unless the column is also defined as `NOT NULL`. For DB2, make the following changes to the code:

```
CREATE TABLE MyUniqueTable
(
    Column1 int,
    Column2 varchar(20) NOT NULL UNIQUE,
    Column3 varchar(12) NOT NULL UNIQUE
);
```

Execute the SQL that's right for your database system, and then try to insert the following values with the SQL shown here:

```
INSERT INTO MyUniqueTable(Column1,Column2, Column3)
VALUES (123,'ABC', 'DEF');

INSERT INTO MyUniqueTable(Column1,Column2, Column3)
VALUES (123,'XYZ', 'DEF');
```

You should find that the first `INSERT INTO` executes just fine and adds a new record, but the second `INSERT INTO` tries to add a second record where the value of `Column3` is `DEF`, violating the `UNIQUE` constraint that you added to the column definition. An error message should appear, which says something like `Violation of UNIQUE KEY constraint 'UQ__MyUniqueTable__3A81B327'. Cannot insert duplicate key in object 'MyUniqueTable'. The statement has been terminated.`

The error message's text varies between database systems, but the message is the same: If the value is not unique, you can't add it.

Setting the `UNIQUE` constraint by simply adding `UNIQUE` after a column is nice and easy, but there's another way that has two advantages. In this alternative, you add the constraint at the end of the column listing in the table definition, which allows you to specify that two or more columns must in combination be unique. The other advantage is that you can give the constraint a name and you can delete the constraint using SQL. You're done with `MyUniqueTable`, so you can delete it:

```
DROP TABLE MyUniqueTable;
```

The following code creates a new table, `AnotherTable`, and adds a unique constraint called `MyUniqueConstraint`. Remember that it needs to be changed on IBM's DB2. This constraint specifies that `Column2` and `Column3` must in combination be unique:

```
CREATE TABLE AnotherTable
(
    Column1 int,
    Column2 varchar(20),
    Column3 varchar(12),
    CONSTRAINT MyUniqueConstraint UNIQUE(Column2, Column3)
);
```

On DB2 you need to ensure that the column doesn't accept NULLs:

```
CREATE TABLE AnotherTable
(
    Column1 int,
    Column2 varchar(20) NOT NULL,
    Column3 varchar(12) NOT NULL,
    CONSTRAINT MyUniqueConstraint UNIQUE(Column2, Column3)
);
```

Try running the preceding SQL for your database system to create the table and then try inserting data with the following `INSERT INTO` statements:

```
INSERT INTO AnotherTable (Column1, Column2, Column3)
VALUES (1, 'ABC', 'DEF');

INSERT INTO AnotherTable (Column1, Column2, Column3)
VALUES (2, 'ABC', 'XYZ');

INSERT INTO AnotherTable (Column1, Column2, Column3)
VALUES (3, 'DEF', 'XYZ');

INSERT INTO AnotherTable (Column1, Column2, Column3)
VALUES (4, 'ABC', 'DEF');
```

The first three `INSERT INTO` statements execute and insert the records. Even though the first two `INSERT` statements add records where `Column2` has the value `ABC`, the record is allowed because the value is unique in combination with `Column3`. The final `INSERT` statement fails because the first `INSERT` statement already entered the combination of `ABC` for `Column2` and `DEF` for `Column3` into the table. Therefore, the combination is no longer unique and violates the constraint `MyUniqueConstraint`, which states that the combination of `Column2` and `Column3` must be unique. Delete `AnotherTable`:

```
DROP TABLE AnotherTable;
```

Chapter 4

You can add more than one constraint to a table, so long as each constraint has a different name and is separated by a comma:

```
CREATE TABLE AnotherTable
(
    Column1 int,
    Column2 varchar(20),
    Column3 varchar(12),
    CONSTRAINT MyUniqueConstraint UNIQUE(Column2, Column3),
    CONSTRAINT AnotherConstraint UNIQUE(Column1, Column3)
);
```

If you're using DB2, you would write the following statement:

```
CREATE TABLE AnotherTable
(
    Column1 int NOT NULL,
    Column2 varchar(20) NOT NULL,
    Column3 varchar(12) NOT NULL,
    CONSTRAINT MyUniqueConstraint UNIQUE(Column2, Column3),
    CONSTRAINT AnotherConstraint UNIQUE(Column1, Column3)
);
```

You can delete AnotherTable from the database:

```
DROP TABLE AnotherTable;
```

So far you've learned how to add a `UNIQUE` constraint at the time of a table's creation. However, using the `ALTER TABLE` statement, you can add and remove a `UNIQUE` constraint after the table's creation. Chapter 1 discussed the `ALTER TABLE` statement in relation to adding and removing columns in a table. To add a constraint, you specify which table to alter and then state that you want to `ADD` a constraint. The SQL code required to define a constraint with an `ALTER TABLE` statement is identical to how you create a constraint at table creation.

The following code demonstrates how to add a constraint called `MyUniqueConstraint` to a table called `YetAnotherTable`:

```
CREATE TABLE YetAnotherTable
(
    Column1 int,
    Column2 varchar(20),
    Column3 varchar(12)
);

ALTER TABLE YetAnotherTable
ADD CONSTRAINT MyUniqueConstraint UNIQUE(Column2, Column3);
```

If you're using DB2, the code is as follows:

```
CREATE TABLE YetAnotherTable
(
    Column1 int,
    Column2 varchar(20) NOT NULL,
    Column3 varchar(12) NOT NULL
);

ALTER TABLE YetAnotherTable
ADD CONSTRAINT MyUniqueConstraint UNIQUE(Column2, Column3);
```

This constraint is identical to the one created earlier for AnotherTable. Again, you can add more than one constraint to a table, just as you did with the constraint definition when you defined the table.

Use `ALTER TABLE` to delete the constraint, and simply drop it as shown below, unless you're using MySQL:

```
ALTER TABLE YetAnotherTable
DROP CONSTRAINT MyUniqueConstraint;
```

If you're using MySQL, the code is as follows:

```
ALTER TABLE YetAnotherTable
DROP INDEX MyUniqueConstraint;
```

You can also use the preceding code to drop constraints created at the time of a table's creation, as long as the constraint has a name.

The table YetAnotherTable can be deleted:

```
DROP TABLE YetAnotherTable;
```

CHECK Constraint

The `CHECK` constraint enables a condition to check the value being entered into a record. If the condition evaluates to `false`, the record violates the constraint and isn't entered into the table. For example, allowing a column storing an age to contain a negative value doesn't make sense. After all, how many people do you know who are minus 35 years old? The `CHECK` condition can be any valid SQL condition, similar to those found in a `WHERE` clause, although SQL can check only the table into which you insert a record. Note that although MS Access supports the `CHECK` constraint, its implementation is outside the scope of this book, and the code examples can't be executed from the main MS Access query designer.

You can add a `CHECK` constraint either at the time of a table's creation or when you alter a table. The following Try It Out shows you how to use the `CHECK` constraint.

Try It Out Adding a CHECK Constraint

1. Using the following SQL, add a CHECK clause to the Age column:

```
CREATE TABLE NamesAges
(
    Name varchar(50),
    Age int CHECK (Age >= 0)
);
```

2. Execute the table creation SQL from Step 1, and then execute the following INSERT INTO statements:

```
INSERT INTO NamesAges (Name, Age)
VALUES ('Jim', 30);

INSERT INTO NamesAges (Name)
VALUES ('Jane');

INSERT INTO NamesAges (Name, Age)
VALUES ('Will', -22);
```

How It Works

Whenever you add a record to the table, the condition on the Age column must evaluate to `true` or `unknown` for the database to accept the record. If Age is 22, the condition is `true` and the database inserts the record. If you try to insert -22, the record violates the CHECK clause and is not inserted. If you insert no value, the clause evaluates to `unknown` and is valid.

The first INSERT INTO statement works because the CHECK condition evaluates to `true`. Indeed, 30 is greater than or equal to zero.

The second INSERT INTO is successful because the CHECK condition evaluates to `unknown`, or `NULL`, because `NULL >= 0` is `unknown`.

The final INSERT INTO fails because the CHECK condition evaluates to `false`: `-22 >= 0` is `false` because Age values can't be negative numbers.

Given that `NULL` data types are always considered valid, you can prevent them by adding a `NOT NULL` constraint. Drop the NamesAges table:

```
DROP TABLE NamesAges;
```

Now that you're through with the Try It Out, consider the following table definition:

```
CREATE TABLE NamesAges
(
    Name varchar(50),
    Age int NOT NULL CHECK (Age >= 0)
);
```

The NOT NULL constraint prevents this INSERT INTO statement from being considered valid:

```
INSERT INTO NamesAges (Name)
VALUES ('Jane');
```

The statement is considered invalid because only the Name column is having data inserted into it; the Age column is left off, so NULL is inserted, causing an error because it conflicts with the NOT NULL constraint. Drop the table:

```
DROP TABLE NamesAges;
```

A problem with adding a CHECK clause to an individual column definition is that the condition can check only that column. For example, the CHECK clause added to AvgMonthlyWage below is invalid and causes the DBMS to throw an error because it contains the column HourlyRate in the condition:

```
CREATE TABLE Employee
(
    EmployeeName varchar(50),
    AvgMonthlyWage decimal(12,2) CHECK (AvgMonthlyWage > HourlyRate),
    HourlyRate decimal(12,2)
);
```

If you want your CHECK condition clause (the clause defined following the CHECK statement) to include multiple columns from the table, you need to define it at the end of the column definitions. Rewrite the preceding SQL like this:

```
CREATE TABLE Employee
(
    EmployeeName varchar(50),
    AvgMonthlyWage decimal(12,2),
    HourlyRate decimal(12,2),
    CONSTRAINT HourlyLess CHECK (AvgMonthlyWage > HourlyRate)
);
```

This SQL creates a table constraint called HourlyLess, which contains a condition that checks that AvgMonthlyWage is greater than HourlyRate.

You can see that adding a CHECK constraint is nearly identical to adding a UNIQUE constraint. Another similarity is the way in which you add a CHECK constraint after you create a table. As with the UNIQUE constraint, you use the ALTER TABLE statement along with ADD CONSTRAINT, the constraint's name, the type of constraint, and in this case, the CHECK constraint's condition.

The following code adds an HourlyLess constraint to the Employee table, though its name conflicts with the constraint of the same name defined when the table was created:

```
ALTER TABLE Employee
ADD CONSTRAINT HourlyLess CHECK (AvgMonthlyWage > HourlyRate);
```

Chapter 4

Obviously, if the table included a constraint called `HourlyLess`, which it does in this case, the DBMS will throw an error. To delete an existing constraint, simply use the `DROP` statement as shown below. Unfortunately, this won't work on MySQL:

```
ALTER TABLE Employee
DROP CONSTRAINT HourlyLess;
```

With the old constraint gone, you can now run the `ALTER TABLE` code and add the constraint with the name `HourlyLess`. You can now drop the `Employee` table:

```
DROP TABLE Employee;
```

The constraint covered in the next section is the very important `PRIMARY KEY` constraint.

Primary Key and PRIMARY KEY Constraint

Of all the constraints, `PRIMARY KEY` is the most important and most commonly used. In fact, every table should have a primary key because first normal form requires it. A primary key provides a link between tables. For example, `MemberId` is the primary key in the Film Club database's `MemberDetails` table and is linked to the `FavCategory` and `Attendance` tables.

In order for the relationship to work, the primary key must uniquely identify a record, which means that you can include only unique values and no `NULL`s. In fact, the `PRIMARY KEY` constraint is essentially a combination of the `UNIQUE` and `NOT NULL` constraints.

Whereas you can have more than one `UNIQUE` or `CHECK` constraint on a table, only one `PRIMARY KEY` constraint per table is allowed.

Given its importance, you should decide on a primary key at the database design state, before you create any tables in the database. Creating the `PRIMARY KEY` constraint is usually easiest when writing the table creation SQL. The format is very similar to the `UNIQUE` and `CHECK` constraints. You can specify the primary key as either a single column or more than one column. To specify a single column, simply insert `PRIMARY KEY` after its definition, like this:

```
CREATE TABLE HolidayBookings
(
    CustomerId int PRIMARY KEY,
    BookingId int,
    Destination varchar(50)
);
```

Note that with IBM DB2, the primary key column must also be defined as `NOT NULL`. The following code works on the other database systems but isn't strictly necessary:

```
CREATE TABLE HolidayBookings
(
    CustomerId int NOT NULL PRIMARY KEY,
    BookingId int,
    Destination varchar(50)
);
```


In the preceding code, `CustomerId` is the primary key. Alternatively, more than one column can act as the primary key, in which case you need to add the constraint at the end of the table, after the column definitions.

The following Try It Out shows you how to use the `PRIMARY KEY` constraint.

Try It Out Establishing a PRIMARY KEY Constraint

1. Execute the following SQL to create the `MoreHolidayBookings` table:

```
CREATE TABLE MoreHolidayBookings
(
    CustomerId int NOT NULL,
    BookingId int NOT NULL,
    Destination varchar(50),
    CONSTRAINT booking_pk PRIMARY KEY (CustomerId, BookingId)
);
```

2. Now try executing the following SQL to insert data into the `MoreHolidayBookings` table:

```
INSERT INTO MoreHolidayBookings (CustomerId, BookingId, Destination)
VALUES (1,1,'Hawaii');

INSERT INTO MoreHolidayBookings (CustomerId, BookingId, Destination)
VALUES (1,2,'Canada');

INSERT INTO MoreHolidayBookings (CustomerId, BookingId, Destination)
VALUES (2,2,'England');

INSERT INTO MoreHolidayBookings (CustomerId, BookingId, Destination)
VALUES (1,1,'New Zealand');

INSERT INTO MoreHolidayBookings (CustomerId, Destination)
VALUES (3,'Mexico');
```

How It Works

In Step 1, the `booking_pk` constraint is a `PRIMARY KEY` constraint, and the columns `CustomerId` and `BookingId` are the primary key columns. Remember that the `PRIMARY KEY` constraint is a combination of the `UNIQUE` and `NOT NULL` constraints, which means that the `CustomerId` and `BookingId` columns cannot contain `NULL` values. They must also be unique in combination. Thus, you can have the same value a number of times in `CustomerId` as long as `BookingId` is different each time, and vice versa. Note that the `NOT NULL` constraint following the primary key columns is strictly necessary only with DB2, but the code works fine on the other database systems even though `NOT NULL` isn't needed.

The first three `INSERT INTO` statements work fine, but the fourth one doesn't because it tries to insert the same combination of values for `CustomerId` and `BookingId`, which the first `INSERT INTO` statement already inserted. Because the primary key constraint doesn't allow duplicates, you receive an error message.

The final `INSERT INTO` also fails because it doesn't assign a value for `BookingId` when inserting a new record, so `BookingId` is `NULL`. Likewise, you receive an error message because the primary key constraint does not allow `NULL` values.

Chapter 4

Drop the MoreHolidayBookings table:

```
DROP TABLE MoreHolidayBookings;
```

Returning to the regular discussion, although good practice is to create the primary key upon table creation, it's also possible to add a primary key to an existing table. The method of doing so is very similar to adding a `UNIQUE` constraint: you use the `ALTER TABLE` statement. However, there is one proviso: The columns forming part of the primary key must already contain the `NOT NULL` constraint. If you try to add a primary key to a column that allows `NULLS`, you receive an error message.

With that in mind, if the `CustomerId` and `BookingId` columns of the `MoreHolidayBookings` table already defined the `NOT NULL` constraint when the table was created:

```
CREATE TABLE MoreHolidayBookings
(
    CustomerId int NOT NULL,
    BookingId int NOT NULL,
    Destination varchar(50)
);
```

Then with the `NOT NULL` constraint already defined, you can add the `PRIMARY KEY` constraint:

```
ALTER TABLE MoreHolidayBookings
ADD CONSTRAINT more_holiday_pk PRIMARY KEY (CustomerId, BookingId);
```

The constraint is called `more_holiday_pk` and it defines `CustomerId` and `BookingId` as the primary key columns. If the `NOT NULL` constraint weren't defined prior to altering the table, you would receive an error.

If you have an existing table without a column or columns to which you want to add a primary key, but that weren't created with the `NOT NULL` constraint, then you must add the `NOT NULL` constraint, as shown earlier in the chapter. As an alternative to adding the `NOT NULL` constraint, simply save the data in a temporary table and then re-create the table with a `PRIMARY KEY` constraint. Your RDBMS may have other ways around the problem that are specific to that system.

Finally, deleting a `PRIMARY KEY` constraint is the same as deleting the `UNIQUE` and `CHECK` constraints. Except on MySQL, use an `ALTER TABLE` statement coupled with a `DROP` statement:

```
ALTER TABLE MoreHolidayBookings
DROP CONSTRAINT more_holiday_pk;
```

On MySQL, the code is as follows:

```
ALTER TABLE MoreHolidayBookings
DROP PRIMARY KEY;
```

The `MoreHolidayBookings` table is no longer needed, so you should drop it:

```
DROP TABLE MoreHolidayBookings;
```

Foreign Key

Foreign keys are columns that refer to primary keys in another table. Primary and foreign keys create relations between data in different tables. Chapter 1 very briefly covered foreign keys, and the Film Club database was designed with a number of tables with primary and foreign keys. When you designed the Film Club database, you didn't set up any `PRIMARY KEY` constraints, but they link tables nevertheless. For example, the `MemberDetails` table contains all the personal data about a member, their name, address, and so on. It also has a primary key column, `MemberId`. Other tables, such as `Attendance` and `FavCategory`, also contain data about members, but rather than repeat personal details about the member, you created a foreign key, `MemberId`, that links a record in one table to a record containing a member's personal details in the `MemberDetails` table. Figure 4-1 illustrates this relationship.

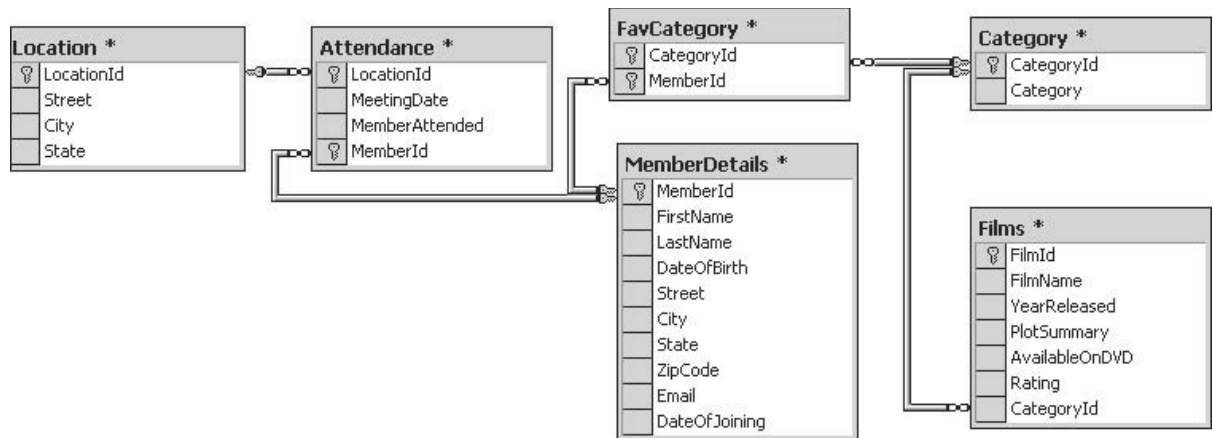


Figure 4-1

Each of the lines linking the tables represents a relationship between the tables. For example, `Attendance` doesn't store all the location details; it simply stores the primary key value from the `Location` table for that particular location. `LocationId` in `Attendance` is therefore a foreign key, and `LocationId` in the `Location` table contains the primary key to which the foreign key refers.

When discussing the primary key, you learned how important it is for it not to contain `NULLs` and for it to contain unique values; otherwise you lose the relationship, and queries return invalid data. The `PRIMARY KEY` constraint helps ensure that duplicate primary key values never occur and prevents `NULLs` being entered. But so far there's nothing stopping someone from entering a value in a foreign key table that doesn't have a matching value in the primary key table. For example, in the Film Club database, the `Location` table's primary key column is `LocationId`. Currently the `Location` table holds only three records, with the values in the records' primary key column having the values 1, 2, and 3:

LocationId	Street	City	State
1	Main Street	Orange Town	New State
2	Winding Road	Windy Village	Golden State
3	Tiny Terrace	Big City	Mega State

Chapter 4

If you want to return a results set detailing meeting dates, the `MemberId` of each member who attended, and the street, city, and state of the meeting, you need to get the information from the `MemberDetails` and `Location` tables. To ensure that the data is linked — that the member who attended and the location where they attended are correctly matched in each row — you need to join the `Location` and `Attendance` tables with an `INNER JOIN`:

```
SELECT MemberId, MeetingDate, Street, City, State
FROM Location INNER JOIN Attendance
ON Attendance.LocationId = Location.LocationId;
```

Doing so returns the results in the following table:

MemberId	MeetingDate	Street	City	State
1	2004-01-01	Winding Road	Windy Village	Golden State
4	2004-01-01	Winding Road	Windy Village	Golden State
5	2004-01-01	Winding Road	Windy Village	Golden State
6	2004-01-01	Winding Road	Windy Village	Golden State
1	2004-03-01	Main Street	Orange Town	New State
4	2004-03-01	Main Street	Orange Town	New State
5	2004-03-01	Main Street	Orange Town	New State
6	2004-03-01	Main Street	Orange Town	New State

The following SQL inserts a record into the `Attendance` table where the `LocationId` has no matching `LocationId` in the `Location` table:

```
INSERT INTO Attendance (LocationId, MeetingDate, MemberAttended, MemberId)
VALUES (99, '2005-12-01', 'Y', 3);
```

Running the preceding query again produces the results in the following table:

MemberId	MeetingDate	Street	City	State
1	2004-01-01	Winding Road	Windy Village	Golden State
4	2004-01-01	Winding Road	Windy Village	Golden State
5	2004-01-01	Winding Road	Windy Village	Golden State
6	2004-01-01	Winding Road	Windy Village	Golden State
1	2004-03-01	Main Street	Orange Town	New State

MemberId	MeetingDate	Street	City	State
4	2004-03-01	Main Street	Orange Town	New State
5	2004-03-01	Main Street	Orange Town	New State
6	2004-03-01	Main Street	Orange Town	New State

Notice the difference? That's right, there is no difference! The new attendance record doesn't show up in the results because the added data is invalid, and the relationship between the Location and Attendance tables has been broken for that record. How can you prevent the relationship between tables from breaking? SQL has the `FOREIGN KEY` constraint for this very purpose. It allows you to specify when a column in a table is a foreign key from another table—when a column in one table is really just a way of referencing rows in another table. For example, you can specify that LocationId in the Attendance table is a foreign key and that it's dependent on the primary key value existing in the Location table's LocationId column.

The basic syntax to create a foreign key is shown in the following code:

```
ALTER TABLE name_of_table_to_add_foreign_key
ADD CONSTRAINT name_of_foreign_key
FOREIGN KEY (name_of_column_that_is_foreign_key_column)
REFERENCES name_of_table_that_is_referenced(name_of_column_being_referenced)
```

The following Try It Out walks you through the process of creating a `FOREIGN KEY` constraint. Unfortunately, the code won't work on IBM's DB2, as before a primary key constraint can be added, the column must have the `NOT NULL` constraint. This can be added in DB2 only at the time the table is created, not after it. Although a `CHECK` clause can be added to stop NULLs, it's not the same as a `NOT NULL` constraint. The only option is to delete a table and re-create it, this time with the appropriate columns with a `NOT NULL` constraint. However, deleting and re-creating the table will result in the loss of the data in the dropped table, unless you transfer it to another table, something not covered until Chapter 5. The code is included here, but it is more fully explained in the next chapter.

Try It Out Creating a FOREIGN KEY Constraint

1. First, delete the invalid record in the Attendance table:

```
DELETE FROM Attendance
WHERE LocationId = 99;
```

2. Add the `NOT NULL` constraint:

```
ALTER TABLE Location
ALTER COLUMN LocationId int NOT NULL;
```

If you're using Oracle or MySQL, the SQL code is as follows:

```
ALTER TABLE Location
MODIFY LocationId int NOT NULL;
```

Chapter 4

If you're using IBM DB2, you need to use the following code:

```
CREATE TABLE TempLocation
(
    LocationId integer NOT NULL,
    Street varchar(100),
    City varchar(75),
    State varchar(75)
);

INSERT INTO TempLocation SELECT * FROM Location;

DROP TABLE Location;

CREATE TABLE Location
(
    LocationId integer NOT NULL,
    Street varchar(100),
    City varchar(75),
    State varchar(75)
);

INSERT INTO Location SELECT * FROM TempLocation;
DROP TABLE TempLocation;
```

You must add a `NOT NULL` constraint before any attempt to add a primary key, lest you receive an error message.

3. Next, add the primary key:

```
ALTER TABLE Location
ADD CONSTRAINT locationid_pk PRIMARY KEY (LocationId);
```

4. Finally, add the FOREIGN KEY constraint:

```
ALTER TABLE Attendance
ADD CONSTRAINT locationid_fk
FOREIGN KEY (LocationId)
REFERENCES Location(LocationId);
```

How It Works

In the first step, you simply deleted the erroneous record in the Attendance table, the record with a LocationId of 99.

Next, you specified which table you want to alter, much as with the other constraints. Before you can create a FOREIGN KEY constraint, however, you need to ensure that the column you specify as the primary key is actually subject to a PRIMARY KEY constraint. However, you can add a PRIMARY KEY constraint to only a column that has a NOT NULL constraint, and the LocationId column has no such constraint. Therefore, the second step added the ADD CONSTRAINT followed by the constraint name and set the primary key. The IBM DB2 syntax is particularly convoluted. It creates a temporary table identical in structure to the Location table. Then data from the existing Location table is copied to this temporary table. After that, the Location table is dropped and then re-created, but with a NOT NULL constraint on the LocationId column, the one to which you want to add a primary key. Finally, the temporary table is dropped.

The third step had you create the primary key on the Location table's LocationId column.

The third line of the fourth step stated the type of constraint, `FOREIGN KEY`, and the brackets following contain all the columns that in combination make up the foreign key. On the final line of code, the `REFERENCES` statement established which table the foreign key references. Following that is the name of the table that holds the primary key and, in brackets, a list of the primary key columns that the foreign key references.

In the preceding example, the primary key, and therefore the matching foreign key, consists of just one column; however, they can consist of more than one column. For example, three columns form the primary and foreign keys in the following SQL:

```
ALTER TABLE SomeTable
ADD CONSTRAINT SomeTable_fk1
FOREIGN KEY (EmployeeName, EmployeeId, MemberShipId)
REFERENCES SomePrimaryKeyTable(EmployeeName, EmployeeId, MemberShipId)
```

Now that the `FOREIGN KEY` constraint is in place, try the `INSERT INTO` statement from earlier:

```
INSERT INTO Attendance (LocationId, MeetingDate, MemberAttended, MemberId)
VALUES (99, '2005-12-01', 'Y', 3);
```

It isn't entered into the database, and you receive an error message.

So far, you've learned how to add a `FOREIGN KEY` constraint to an existing table, but it can be defined at the time the table is created, in a similar manner to the `CHECK` and `UNIQUE` constraints. As demonstrated in the following code, you add the constraint definition at the end of the table definition, just after the column definitions:

```
CREATE TABLE Attendance
(
    LocationId integer,
    MeetingDate date,
    MemberAttended char(1),
    MemberId integer,
    CONSTRAINT SomeTable_fk1
    FOREIGN KEY (LocationId)
    REFERENCES Location(LocationId)
);
```

Those are all the constraints covered in this book. The next section examines how to speed up results using an index.

Speeding Up Results with Indexes

Indexes are special lookup tables that the database search engine can use to speed up data retrieval. They also organize the way in which the database stores data. A good real-world example is a book, much like this one, that has an index at the back that helps you find where the important information is. It saves a lot of flicking through the pages randomly trying to find the topic you want. Given that an

Chapter 4

index helps speed up `SELECT` queries and `WHERE` clauses, why not always have one? First of all, while it speeds up data retrieval, it slows down data input, with `UPDATE` and `INSERT` statements, for example. Additionally, it adds to a database's size, albeit not massively, though it is a consideration. Using indexes is good practice, but it is best done judiciously — for example, using them to help your most common queries.

Creating an index involves the `CREATE INDEX` statement, which allows you to name the index, to specify the table and which column or columns to index, and to indicate whether the index is in ascending or descending order. Indexes can also be unique, similar to the `UNIQUE` constraint, in that the index prevents duplicate entries in the column or combination of columns on which there's an index. The basic format of the statement is as follows:

```
CREATE INDEX <index_name>
ON <table_name> (<column_names>)
```

The following code adds an index called `member_name_index` on the `MemberDetails` table, and it indexes the `FirstName` and `LastName` columns:

```
CREATE INDEX member_name_index
ON MemberDetails (FirstName, LastName);
```

If you execute the following `SELECT` statement, you notice something interesting:

```
SELECT FirstName, LastName
FROM MemberDetails;
```

The results, shown in the following table, appear in ascending order, by first name and then last name, even though you did not add an `ORDER BY` clause:

FirstName	LastName
Catherine	Hawthorn
Doris	Night
Jack	Johnson
Jamie	Hills
Jenny	Jones
John	Jackson
John	Jones
Katie	Smith
Seymour	Botts
Steve	Gee
Stuart	Dales
Susie	Simons
William	Doors

The results are ordered because, by default, the index orders results in ascending order. The default order is based on the order in which you list the columns in the index definition's `SELECT` statement. To delete the index, you need to use the `DROP INDEX` statement, specifying the index name in the statement. Note that in some RDBMSs, such as MS SQL Server, you need to specify the table name in addition to the index name. So, the SQL to drop the index just created in MS SQL Server is as follows:

```
DROP INDEX MemberDetails.member_name_idx;
```

In IBM DB2 and Oracle, the `DROP INDEX` statement simply requires the index name without the table name prefixed:

```
DROP INDEX member_name_idx;
```

In MySQL, the code to drop the index is as follows:

```
ALTER TABLE MemberDetails
DROP INDEX member_name_idx;
```

MS Access has yet another way of dropping the index:

```
DROP INDEX member_name_idx ON MemberDetails;
```

After dropping the index, run the same `SELECT` statement:

```
SELECT FirstName, LastName
FROM MemberDetails;
```

You find that the results are no longer necessarily in order:

FirstName	LastName
Katie	Smith
Susie	Simons
John	Jackson
Steve	Gee
John	Jones
Jenny	Jones
Jack	Johnson
Seymour	Botts
Jamie	Hills
Stuart	Dales
William	Doors
Doris	Night
Catherine	Hawthorn

Chapter 4

The results you get may be in a slightly different order, depending on your DBMS. When results are not ordered, there's no real guarantee of what the order might be.

You can set two other options when creating an index. The first, the `UNIQUE` option, prevents duplicate values from being entered and works very much like the `UNIQUE` constraint. The second option determines the column order. Recall that the default results order is ascending, but you can also order results in descending order.

The following SQL creates a unique index that orders results by last name in descending order and then by first name. Using the `DESC` keyword, execute the following SQL:

```
CREATE UNIQUE INDEX member_name_idx
ON MemberDetails (LastName DESC, FirstName);
```

After executing the preceding code, execute this query:

```
SELECT LastName, FirstName
FROM MemberDetails;
```

Query results are provided in the following table:

LastName	FirstName
Botts	Seymour
Dales	Stuart
Doors	William
Gee	Steve
Hawthorn	Catherine
Hills	Jamie
Jackson	John
Johnson	Jack
Jones	Jenny
Jones	John
Night	Doris
Simons	Susie
Smith	Katie

You can see that `LastName` now goes in order from the lowest to the highest — from *a* to *z* in the alphabet.

The index is no longer needed, so delete it with the following code. If using MS SQL Server, write the following `DROP INDEX` statement:

```
DROP INDEX MemberDetails.member_name_idx;
```

In IBM DB2 and Oracle, the `DROP INDEX` statement simply requires the index name without the table name prefixed:

```
DROP INDEX member_name_idx;
```

In MySQL, the code to drop the index is as follows:

```
ALTER TABLE MemberDetails  
DROP INDEX member_name_idx;
```

MS Access has yet another way of dropping the index:

```
DROP INDEX member_name_idx ON MemberDetails;
```

You've learned a lot of useful stuff for improving the design and efficiency of your databases. The next section takes that knowledge and applies it to the Film Club database.

Improving the Design of the Film Club Database

This section revisits the Film Club database, taking into account the topics discussed in this chapter: normalization, ensuring data integrity with constraints, and using indexes to speed up data retrieval.

The discussion begins with an examination of the database structure and then turns to constraints and indexes.

Reexamining the Film Club Database Structure

Generally speaking, the database is in third normal form, with the exception of the MemberDetails table. The Street, City, and State columns are all transitively dependent on the ZipCode column. Therefore, the MemberDetails table is still in second normal form. The question now, however, is one of common sense, because the amount of duplication reduced by going to third normal form is going to be quite small. It's unlikely that many people from the same street would join the club, and with a small database, storage size is not usually an issue. In addition, changing the table to comply with third normal form involves creating another table, and as a result any `SELECT` queries require another `JOIN` statement, which impinges on database efficiency. Therefore, in this instance, the database is fine as it is, and moving the MemberDetails table to third normal form is probably unnecessary and only adds complexity without really saving any storage space.

However, one area of the database design needs altering: the Attendance table. Consider the following query:

```
SELECT * FROM Attendance;
```

Chapter 4

The results of the query are shown in the following table:

MeetingDate	MemberAttended	MemberId	LocationId
2004-01-01	Y	1	2
2004-01-01	N	4	2
2004-01-01	Y	5	2
2004-01-01	Y	6	2
2004-03-01	Y	1	1
2004-03-01	Y	4	1
2004-03-01	N	5	1
2004-03-01	N	6	1

For each meeting, the Attendance table stores each film club member and their attendance status. Rather than store a record regardless of whether a member attended or not, you can save space by storing a record only if a member actually attends a meeting. Getting a list of members who attended a meeting is simple enough. The SQL to find out who didn't attend, however, is a little more involved and is covered in greater detail in Chapter 7.

To make the changes, delete all the records where the member didn't attend, that is, where MemberAttended equals N. Then delete the MemberAttended column.

Use this SQL to delete all the records where MemberAttended equals N:

```
DELETE FROM Attendance
WHERE MemberAttended = 'N';
```

Once you delete those records, you can drop the whole column:

```
ALTER TABLE Attendance
DROP COLUMN MemberAttended;
```

Dropping the column deletes all the data stored in that column for all rows. Unfortunately, IBM's DB2 doesn't support dropping a column. The only option, as you saw earlier, is to delete the whole table and re-create it from scratch, having first saved the data:

```
CREATE TABLE TempAttendance
(
    MeetingDate date,
    LocationId integer,
    MemberId integer
);

INSERT INTO TempAttendance (MeetingDate, LocationId, MemberId)
SELECT MeetingDate, LocationId, MemberId FROM Attendance
```

```
WHERE MemberAttended = 'Y';

DROP TABLE Attendance;

CREATE TABLE Attendance
(
    MeetingDate date NOT NULL,
    LocationId integer NOT NULL,
    MemberId integer NOT NULL
);

INSERT INTO Attendance (MeetingDate, LocationId, MemberId)
SELECT MeetingDate, LocationId, MemberId FROM Attendance;
DROP TABLE TempAttendance;
```

As in the earlier example, the data from the table that you need to change is saved in a temporary table. The `MemberAttended` column is being deleted, so there's no need to save data from that. The original table is deleted and then re-created, but without the `MemberAttended` column. The data is then copied back from the `TempAttendance` table, which is then dropped.

The changes made to the `Attendance` table have an additional consequence as far as `SELECT` queries go. Displaying all members who attended is now easier. You don't need to include a `WHERE` clause specifying that `MemberAttended` equals `Y`; a basic `SELECT * FROM Attendance` statement now suffices. However, finding out lists of members who didn't attend meetings is harder and involves subqueries, a topic covered in Chapter 7.

Often, however, the downside to more efficient data storage is more complex SQL queries, covered in more detail in Chapter 7. The next section deals with ensuring data validation in your more efficient database.

Improving Data Validation and Efficiency

This section employs the various constraints discussed earlier to help reduce the risk of data corruption. A good first step is to enforce the primary keys in each table by adding a `PRIMARY KEY` constraint.

The `Location` table already has a `PRIMARY KEY` constraint, added when you learned about the `FOREIGN KEY` constraint. Begin by adding a `PRIMARY KEY` constraint to the `MemberDetails` table. First, however, you need to add a `NOT NULL` constraint because the `PRIMARY KEY` constraint can't be added to a column that allows `NULL`s:

```
ALTER TABLE MemberDetails
ALTER COLUMN MemberId int NOT NULL;
```

If using Oracle or MySQL, you need to change the code to the following:

```
ALTER TABLE MemberDetails
MODIFY MemberId int NOT NULL;
```

Chapter 4

Remember, IBM DB2 doesn't support adding NOT NULL after the table has been created, so you have to go the long-winded route of copying the data to a temporary table, dropping the MemberDetails table, and then re-creating it with the NOT NULL constraint, as shown in the following code:

```
CREATE TABLE TempMemberDetails
(
    MemberId integer,
    FirstName vargraphic(50),
    LastName vargraphic(50),
    DateOfBirth date,
    Street varchar(100),
    City varchar(75),
    State varchar(75),
    ZipCode varchar(12),
    Email varchar(200),
    DateOfJoining date
);

INSERT INTO TempMemberDetails
SELECT * FROM MemberDetails;

DROP TABLE MemberDetails;

CREATE TABLE MemberDetails
(
    MemberId integer NOT NULL,
    FirstName vargraphic(50),
    LastName vargraphic(50),
    DateOfBirth date,
    Street varchar(100),
    City varchar(75),
    State varchar(75),
    ZipCode varchar(12),
    Email varchar(200),
    DateOfJoining date
);

INSERT INTO MemberDetails
SELECT * FROM TempMemberDetails;

DROP TABLE TempMemberDetails;
```

Having added the NOT NULL constraint, via whichever means your database system requires, now execute the following code, which adds the actual PRIMARY KEY:

```
ALTER TABLE MemberDetails
ADD CONSTRAINT memberdetails_pk PRIMARY KEY (MemberId);
```

Primary keys must be unique and cannot contain NULLs, so if by accident your table contains NULLs or duplicate values for the `MemberId` column, then you need to edit and correct them before the statement can work.

Next, add a `PRIMARY KEY` constraint to the `Films` table. First, you need to add a `NOT NULL` constraint. Use the following code for MS Access and SQL Server:

```
ALTER TABLE Films
ALTER COLUMN FilmId int NOT NULL;
```

If you're using Oracle or MySQL, use the following statement:

```
ALTER TABLE Films
MODIFY FilmId int NOT NULL;
```

Again, with IBM's DB2, you need to add the `NOT NULL` constraint by re-creating the whole table:

```
CREATE TABLE TempFilms
(
    FilmId integer,
    FilmName varchar(100),
    YearReleased integer,
    PlotSummary varchar(2000),
    AvailableOnDVD char(1),
    Rating integer,
    CategoryId integer
);

INSERT INTO TempFilms
SELECT * FROM Films;

DROP TABLE Films;

CREATE TABLE Films
(
    FilmId integer NOT NULL,
    FilmName varchar(100),
    YearReleased integer,
    PlotSummary varchar(2000),
    AvailableOnDVD char(1),
    Rating integer,
    CategoryId integer
);

INSERT INTO Films
SELECT * FROM TempFilms;

DROP TABLE TempFilms;
```

Be sure to check all columns for duplicate entries in order to avoid errors in creating the `PRIMARY KEY` constraint. In this case, the `FilmId` column contains duplicate values. It's surprisingly easy for invalid data to occur, which makes defining a `PRIMARY KEY` constraint essential. To rid the `FilmId` column of duplicates, execute the following three `UPDATE` statements:

Chapter 4

```
UPDATE Films
SET FilmId = 13
WHERE FilmId = 12 AND
FilmName = 'The Good, the Bad, and the Facially Challenged';

UPDATE Films
SET FilmId = 14
WHERE FilmId = 2 AND
FilmName = '15th Late Afternoon';

UPDATE Films
SET FilmId = 15
WHERE FilmId = 2 AND
FilmName = 'Soylent Yellow';
```

Execute the following SQL to create the PRIMARY KEY constraint:

```
ALTER TABLE Films
ADD CONSTRAINT films_pk PRIMARY KEY (FilmId);
```

Next, add a PRIMARY KEY constraint to the Category table. The steps are identical to those described previously. First, add a NOT NULL constraint. If you're using MS Access or SQL Server, the code is as follows:

```
ALTER TABLE Category
ALTER COLUMN CategoryId int NOT NULL;
```

If you're using MySQL or Oracle, use the following statement:

```
ALTER TABLE Category
MODIFY CategoryId int NOT NULL;
```

Once again, with IBM's DB2, you need to re-create the table:

```
CREATE TABLE TempCategory
(
    CategoryId integer,
    Category varchar(100)
);

INSERT INTO TempCategory
SELECT * FROM Category;

DROP TABLE Category;

CREATE TABLE Category
(
    CategoryId integer NOT NULL,
    Category varchar(100)
);

INSERT INTO Category
SELECT * FROM TempCategory;

DROP TABLE TempCategory;
```


Then add the PRIMARY KEY constraint:

```
ALTER TABLE Category
ADD CONSTRAINT category_pk PRIMARY KEY (CategoryId);
```

Just two more tables to go to an efficient database. First, update the FavCategory table. This time, though, the primary key is based on two columns, CategoryId and MemberId, so you need to make sure to add both to the list of columns defining the PRIMARY KEY constraint. First, add the NOT NULL constraint to both columns. In MS Access and SQL Server, the code is as follows:

```
ALTER TABLE FavCategory
ALTER COLUMN CategoryId int NOT NULL;

ALTER TABLE FavCategory
ALTER COLUMN MemberId int NOT NULL;
```

If you're using Oracle or MySQL, the preceding code should read as follows:

```
ALTER TABLE FavCategory
MODIFY CategoryId int NOT NULL;

ALTER TABLE FavCategory
MODIFY MemberId int NOT NULL;
```

In IBM DB2, you should use the following statement:

```
CREATE TABLE TempFavCategory
(
    CategoryId integer,
    MemberId integer
);

INSERT INTO TempFavCategory
SELECT * FROM FavCategory;

DROP TABLE FavCategory;

CREATE TABLE FavCategory
(
    CategoryId integer NOT NULL,
    MemberId integer NOT NULL
);

INSERT INTO FavCategory
SELECT * FROM TempFavCategory;

DROP TABLE TempFavCategory;
```

Now you can add the PRIMARY KEY constraint:

```
ALTER TABLE FavCategory
ADD CONSTRAINT favcategory_pk PRIMARY KEY (CategoryId, MemberId);
```

Chapter 4

Finally, you do the same thing for the Attendance table, beginning with the NOT NULL constraint:

```
ALTER TABLE Attendance
ALTER COLUMN LocationId int NOT NULL;

ALTER TABLE Attendance
ALTER COLUMN MemberId int NOT NULL;
```

Again, for Oracle or MySQL, change the code to the following:

```
ALTER TABLE Attendance
MODIFY LocationId int NOT NULL;

ALTER TABLE Attendance
MODIFY MemberId int NOT NULL;
```

Then you add the PRIMARY KEY constraint:

```
ALTER TABLE Attendance
ADD CONSTRAINT attendance_pk PRIMARY KEY (LocationId, MemberId);
```

Your next objective in improving the database is to prevent columns containing NULL values where necessary. Good practice is to add the NOT NULL constraint when creating the table. For example, if the FilmName column contains no value, it makes the whole record rather pointless, as the name is so essential. On the other hand, having a missing rating is not as much of a problem. Begin by adding a NOT NULL constraint to the FilmName column. In MS Access or SQL Server, use the following code:

```
ALTER TABLE Films
ALTER COLUMN FilmName varchar(100) NOT NULL;
```

In Oracle or MySQL, type this statement:

```
ALTER TABLE Films
MODIFY FilmName varchar(100) NOT NULL;
```

Finally, in IBM DB2, use the following SQL:

```
CREATE TABLE TempFilms
(
    FilmId integer NOT NULL,
    FilmName varchar(100),
    YearReleased integer,
    PlotSummary varchar(2000),
    AvailableOnDVD char(1),
    Rating integer,
    CategoryId integer
);

INSERT INTO TempFilms
SELECT * FROM Films;

DROP TABLE Films;

CREATE TABLE Films
```

```
(
    FilmId integer NOT NULL,
    FilmName varchar(100) NOT NULL,
    YearReleased integer,
    PlotSummary varchar(2000),
    AvailableOnDVD char(1),
    Rating integer,
    CategoryId integer
);

INSERT INTO Films
SELECT * FROM TempFilms;
DROP TABLE TempFilms;
```

Next up is the MemberDetails table. Again, the person's first and last names are pretty much essential to track the person, and therefore these columns should not remain incomplete.

Add the NOT NULL constraints in MS SQL Server or Access with this code:

```
ALTER TABLE MemberDetails
ALTER COLUMN FirstName varchar(50) NOT NULL;

ALTER TABLE MemberDetails
ALTER COLUMN LastName varchar(50) NOT NULL;
```

Change the code in Oracle or MySQL to the following:

```
ALTER TABLE MemberDetails
MODIFY FirstName varchar(50) NOT NULL;

ALTER TABLE MemberDetails
MODIFY LastName varchar(50) NOT NULL;
```

In DB2, use the following statement:

```
CREATE TABLE TempMemberDetails
(
    MemberId integer,
    FirstName vargraphic(50),
    LastName vargraphic(50),
    DateOfBirth date,
    Street varchar(100),
    City varchar(75),
    State varchar(75),
    ZipCode varchar(12),
    Email varchar(200),
    DateOfJoining date
);

INSERT INTO TempMemberDetails
```

Chapter 4

```
SELECT * FROM MemberDetails;

DROP TABLE MemberDetails;

CREATE TABLE MemberDetails
(
    MemberId integer NOT NULL,
    FirstName varchar(50) NOT NULL,
    LastName varchar(50) NOT NULL,
    DateOfBirth date,
    Street varchar(100),
    City varchar(75),
    State varchar(75),
    ZipCode varchar(12),
    Email varchar(200),
    DateOfJoining date
);

INSERT INTO MemberDetails
SELECT * FROM TempMemberDetails;
DROP TABLE TempMemberDetails;
```

Once the NOT NULL constraints are in place, you can add the index again:

```
CREATE UNIQUE INDEX member_name_indx
ON MemberDetails (LastName DESC, FirstName);
```

Finally, define the relationships between tables by adding FOREIGN KEY constraints. Figure 4-1 shows the foreign key links, and the following list reiterates them.

- ☐ The CategoryId column in the Films table is a foreign key linking to the CategoryId column in the Category table.
- ☐ The CategoryId column in the FavCategory table is a foreign key linking to the CategoryId column in the Category table.
- ☐ The MemberId column in the FavCategory table is a foreign key linking to the MemberId column in the MemberDetails table.
- ☐ The LocationId column in the Attendance table is a foreign key linking to the LocationId column in the Location table.
- ☐ The MemberId column in the Attendance table is a foreign key linking to the MemberId column in the MemberDetails table.

In an earlier Try It Out, you defined the relationship between the Attendance and Location tables. The Location table is the primary table to which the Attendance table links via the LocationId column. The Attendance table also links to the MemberDetails table, so you need to define that relationship as well.

If you've already added the constraint in the earlier Try It Out, then don't execute the code again or you'll receive an error. If you didn't execute the code, then the SQL used in an earlier Try It Out to add the constraint was as follows:

```
ALTER TABLE Attendance
ADD CONSTRAINT attend_loc_fk
FOREIGN KEY (LocationId)
REFERENCES Location(LocationId);
```

The Attendance table is the one being altered; the FOREIGN KEY constraint doesn't affect the Location table. This constraint, for example, is named `attend_loc_fk`, a mixture of the two tables involved in the relationship, where `fk` denotes its status as a foreign key. You can name your constraints anything you like; just make sure the name is something that identifies the relationship between tables. Finally, the statement specifies the table and the column or columns being referenced.

The following SQL adds a second FOREIGN KEY constraint to the Attendance table, this time for the relationship between the Attendance and MemberDetails tables:

```
ALTER TABLE Attendance
ADD CONSTRAINT attend_memdet_fk
FOREIGN KEY (MemberId)
REFERENCES MemberDetails(MemberId);
```

The next table that contains a foreign key is the FavCategory table, but defining that relationship is one of the exercise questions at the end of the chapter. That leaves just one more relationship to define: between the Category and Film tables. The Category table is the primary table to which the Films table links. The link is between the CategoryId columns present in both tables. The following code formally defines this relationship:

```
ALTER TABLE Films
ADD CONSTRAINT films_cat_fk
FOREIGN KEY (CategoryId)
REFERENCES Category(CategoryId);
```

That completes the changes to the database. Innumerable ways to improve a database exist, and these depend on how you intend to use the database. Apart from the index on the MemberDetails table's FirstName and LastName fields, you didn't create any new indexes. If in live use you find that some queries take too long to return results, simply review the database structure again and look at changing tables or adding an index to speed things up. Quite a lot of RDBMSs have tools that help monitor performance and tweak settings to ensure the best possible results.

Now that you can design a relatively complex normalized database, complete with constraints and primary and foreign keys, here are some things to keep in mind any time you design or modify a database.

Tips for Designing a Better Database

Keeping the following subjects in mind helps ensure that your database design and updates go smoothly.

- ❑ **Don't design a database that copes with the norm.** Tempting as it is to design a database that covers most situations, doing so is dangerous. The unexpected happens just when you least expect it, so make sure you design your database to cover all situations that could arise, or at least ensure that it can cope with the unusual situations. Even if a client tells you not to worry about the possibility of two people attempting to reserve the same holiday cottage at the same time, assume that it will happen.

- ❑ **Choose meaningful names for tables and fields.** Try to use field and table names that help give an idea of what data they store. For example, the `MemberDetails` table stores members' details, which makes it fairly obvious without further explanation what the table holds. Name tables so that further explanation or looking into the table is unnecessary. The same applies to column names.
- ❑ **Try to keep names simple.** Maybe this seems to contradict the previous point, but it doesn't: Names should be as descriptive as possible, but they shouldn't be overly long or complex. Long names increase the likelihood of errors.
- ❑ **Be consistent in your naming and choice of data type.** To prevent confusion, don't call a field `ZipCode` in one table and `PostalCode` in another if they refer to the same data. Also make sure that both fields are the same data type and can store the same width of data. If you define one as `varchar(12)` in one table and `varchar(8)` in another, you risk truncation if you ever insert from one table into another.
- ❑ **Analyze your data needs on paper first.** It's very tempting when asked to create a database to rush off and start designing on the fly, as it were. However, take time out first to sit down with pen and paper and consider what data needs to be stored and, most importantly, what answers the database is expected to supply. If the person needing the database already operates some other system (for example, a paper-based storage system), take a look at that and use it as your starting point for the data input.
- ❑ **Pick your primary key carefully.** Choose a field that is unlikely to change and preferably one that is a whole-number-based field. The primary key must always be unique. If no field is obvious, then create your own whole-number field for the purpose of creating a unique primary key.
- ❑ **Create an index.** Indexes help speed up searches, so adding them to fields that are regularly used in searches or joins is worthwhile. Indexes are especially worthwhile where you have lots of different values—for example, the `ZipCode` field in the `MemberDetails` table. Including an index is not a good idea, however, if you have only a few values, such as the `MemberAttended` column in the `Attendance` table. Indexes also slow down data entry, something particularly important to note if the column is likely to have lots of inserts or updates.
- ❑ **Add a multicolumn index.** Multicolumn indexes come in particularly handy in fields where users often search more than one column. For example, if you often search for `City` and `State` together, add an index based on both columns.
- ❑ **Avoid using reserved words as table or field names.** Reserved words are words used by the SQL language and are therefore reserved for its use only. For example, words such as *select*, *join*, and *inner* are exclusive to SQL. Although you can sometimes use reserved words by putting square brackets around them, avoiding them altogether is easier.
- ❑ **Consider storage space requirements.** When selecting a field's data type, allow for the maximum storage space likely to be required, and then add a little bit! If you think the greatest number of characters to be stored is probably 8, make your definition 10, or `varchar(10)`. Doing so adds a little bit of a safety net. The same goes with numbers.

Summary

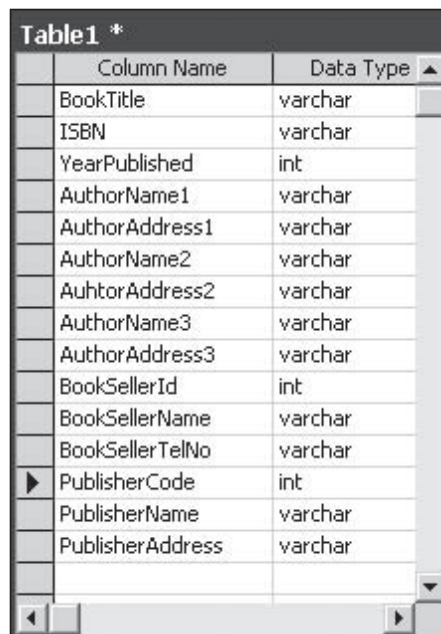
This chapter covered all facets of improving your database designing skills. It looked at normalization and the ways in which it provides steps to follow to minimize redundant data storage, as well as the following topics:

- ❑ You learned the importance of ensuring that data in a database remains valid — for example, ensuring that there are no duplicated values in primary key columns. The mechanism that disallows duplicate data is the constraint, which includes `NOT NULL`, `UNIQUE`, `CHECK`, `PRIMARY KEY`, and `FOREIGN KEY`.
- ❑ You learned how to speed up query results by using indexes.
- ❑ Using the chapter's topics, you improved the Film Club database's design, making it leaner, more efficient, and less likely to fall afoul of invalid data.

The next chapter examines how to manipulate data returned by a query and covers such subjects as arithmetic in SQL.

Exercises

1. Your friend runs a bookstore and uses a simple database to store details of books, authors, and booksellers. She thinks the database might need a bit of a tune-up. Figure 4-2 shows its current structure, which consists of just one table containing two columns: Column Name and Data Type. Using what you learned about normalization, see if you can improve the current structure.



Column Name	Data Type
BookTitle	varchar
ISBN	varchar
YearPublished	int
AuthorName1	varchar
AuthorAddress1	varchar
AuthorName2	varchar
AuthorAddress2	varchar
AuthorName3	varchar
AuthorAddress3	varchar
BookSellerId	int
BookSellerName	varchar
BookSellerTelNo	varchar
PublisherCode	int
PublisherName	varchar
PublisherAddress	varchar

Figure 4-2

2. When improving the Film Club database, you added various constraints as well as primary and foreign keys. You didn't, however, add a `FOREIGN KEY` constraint to the FavCategory table. Create and execute the SQL required to enforce all its relationships with the other tables.