

1

Introduction to SQL

A nice, gentle introductory chapter, this chapter begins by looking at databases in terms of what they are and why and when you want to use them. Then the chapter turns to SQL and discovers how it links in with databases and how it can be useful. After tackling the basics of SQL and how it works in theory, you examine how to use it to create a database. This chapter also walks you through creating the structure of the example database used throughout the book.

By the end of the chapter, you should understand how a database enables you to efficiently organize and retrieve the information you want, as well as how to create a fully functional database, all ready and waiting to accept add data. But before diving headlong into writing lines of SQL code, it's helpful to know a little bit of background about databases.

A Brief History of Databases

Modern databases emerged in the 1960s thanks to research at IBM, among other companies. The research mainly centered around office automation, in particular automating data storage and indexing tasks that previously required a great deal of manual labor. Computing power and storage had become much cheaper, making the use of computers for data indexing and storage a viable solution. A pioneer in the database field was Charles W. Bachman, who received the Turing Award in 1973 for pioneering work in database technology. In 1970, an IBM researcher named Ted Codd published the first article on relational databases.

Although IBM was a leader in database research, Honeywell Information Systems, Inc., released a commercial product in 1976 based on the same principles as the IBM information system, but it was designed and implemented separately from IBM's work.

In the early 1980s, the first database systems built upon the SQL standard appeared from companies such as Oracle, with Oracle Version 2, and later SQL/DS from IBM, as well as a host of other systems from other companies.

Now that you have a brief idea of where databases came from, you can turn to the more practical task of what databases are and why and when to use them.

Identifying Databases

What is a database, you ask?

The Free On-Line Dictionary of Computing (<http://foldoc.doc.ic.ac.uk>) defines a database as “one or more large structured sets of persistent data, usually associated with software to update and query the data. A simple database might be a single file containing many records, each of which contains the same set of fields where each field is a certain fixed width.”

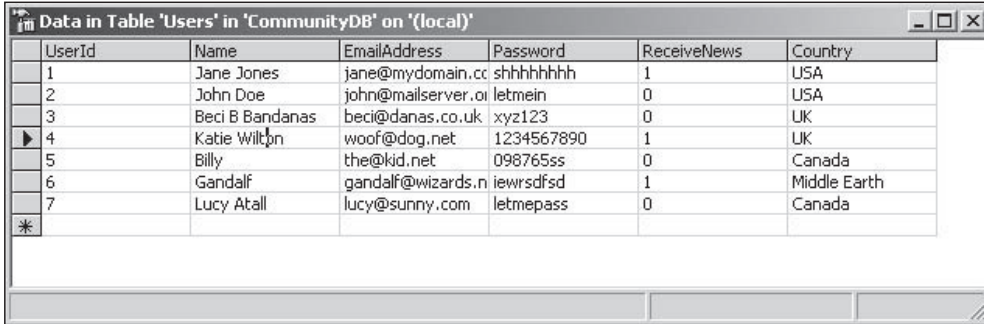
Breaking this definition down into something more manageable, first it says that a database consists of structured sets of data, which means that a database contains collections of data. For example, the database might contain the details of Uncle Bob’s golf scores or data about all the books in a library. You probably wouldn’t want to mix these two collections of data, or else when you want to find data about a book you’d have to look through irrelevant data on golf scores. In short, databases help you organize your data. A database stores its collections of data in *tables*, a concept explored further in Chapter 2.

The definition goes on to say that databases are usually associated with software that allows the data to be updated and queried. Real-life examples of database software include Microsoft’s Access, Oracle’s 10g, IBM’s DB2, MySQL AB’s MySQL, and Microsoft’s SQL Server 2000. Often these programs are referred to as databases, but strictly speaking, they are database management systems (DBMS). A database is the *sets* (collections of related data) grouped into one entity. You could, for example, create an Access database, call it MyDatabase, include various data collections inside that one database, and manage the whole thing with the MS Access software.

Finally, the definition states that, as with the Access database example, a simple database might be just one file with many records with each record broken down into *fields*. But what are records and fields? A field is a single item of data about a specific thing. A thing could be a person, and a single item of data about a person could be their date of birth. Or the thing might be the address of a house and the specific item of data might be its street. Using the book example, the year a book was published is a specific piece of data that can be stored in a field. Another field might be the book’s title; yet another could be the author’s name. For this book, the fields would contain 2005 for the Year Published field, *Beginning SQL* for the Title field, and Paul Wilton and John Colby for the Author field. All these fields refer to one specific thing, a book called *Beginning SQL*. Collectively these fields are known as a *record*. Each book has its own record, and all the records are stored collectively in a database in something called a *table*. A single database can contain one or more tables. If all this information is a bit too much to absorb at once, don’t worry: I’ll be revisiting the concepts of fields and records later in this chapter.

By now, hopefully you get the idea that a database helps you store, organize, and retrieve data. One last thing to mention is the term *relational database*, which is a database containing data organized and linked (related) to each other. All records in a database are organized into tables. Related data, such as details of sales persons, are grouped in one table. You could put the details of cars they have sold in another table and then specify a relationship between which salesperson sold which cars — for example, salesperson X sold car Y on date Z. Figure 1-1 shows a table from the example database. On first glance, you may notice its resemblance to a spreadsheet with rows being your records and columns containing the fields for the records. In Chapter 3 you discover that you really need to think in terms of sets of data.

Most database management systems these days are relational, termed relational database management system (RDBMS). These systems make storing data and returning results easier and more efficient. They allow different questions to be posed of the database — even questions the original designer of the database didn’t expect to be asked.



UserId	Name	EmailAddress	Password	ReceiveNews	Country
1	Jane Jones	jane@mydomain.cc	shhhhhhhh	1	USA
2	John Doe	john@mailserver.oi	letmein	0	USA
3	Beci B Bandanas	beci@danass.co.uk	xyz123	0	UK
4	Katie Wilton	woof@dog.net	1234567890	1	UK
5	Billy	the@kid.net	098765ss	0	Canada
6	Gandalf	gandalf@wizards.n	iewrsdfsd	1	Middle Earth
7	Lucy Atall	lucy@sunny.com	letmepass	0	Canada

Figure 1-1

Why and When to Use a Database

When there are a huge number of alternative ways to store data, why should you trouble yourself creating a database? What advantages does a database hold?

The main advantage is fast and efficient data retrieval. A database helps you to organize your data in a logical manner. Database management systems are fine-tuned to rapidly retrieve the data you want in the way you want it. Databases also enable you to break data into specific parts. Retrieving data from a database is called *querying*. You'll often see the term *SQL query*, which briefly means any SQL code that extracts data from the database. This topic is covered in more depth later in this chapter.

Relational databases have the further advantage of allowing you to specify how different data relates to each other, as you saw in the car sales database example. If you store sales details and salesperson data in related databases, the question "How many cars has salesperson X sold in January?" becomes very easy to answer. If you just shoved all the information into a large text file, you'd find it one enormous task to question, or query, the data and find out specific answers.

Databases also allow you to set up rules that ensure that data remains consistent when you add, update, or delete data. Imagine that your imaginary car sales company has two salespeople named Julie Smith. You can set up a database to ensure that each salesperson has a unique ID, called a unique identifier (so that the Julies don't get mixed up); otherwise, telling who sold which cars would prove impossible. Other data storage systems, such as text files or spreadsheets, don't have these sorts of checks and quite happily allow you to store erroneous data. In later chapters you learn how to set up other rules to limit the risk of data becoming corrupted. For example, you might specify that an employee's social security number must be unique in the database. Or if a car is sold and it's listed as being sold by the employee with an ID of 123, you might add a check to see that full details of employee 123 are held in one of the database tables.

A properly set-up database minimizes data redundancy. Again using the car sales example, you can store all the details of a salesperson just once in the database and then use a unique ID to identify each salesperson. When you have other data that relates to a particular salesperson (for example, which cars they've sold), you can use the unique ID to search for the data. The unique ID is often a number that takes up less storage space than the person's full name.

Databases store raw data — just the facts, so to speak, and no intelligence. A car sales database might contain the make, model, and price of each car, but you wouldn't normally store the average number of cars sold in a month, because you can calculate that from the car sales information, the raw data.

Chapter 1

A spreadsheet, however, may contain processed data, such as averages and statistical analysis. A database simply stores the data and generally leaves data processing to a *front-end* program, or the interface the user sees. Examples of front-end programs include a Web page that draws its data from a database or a program that hooks into the database's data and allows the user to view it.

Sharing data is also much easier using a database. You can share data among a number of users on the same computer or among users on different computers linked via a network or the Internet. If the example car sales company has branches in New York, Washington, and Boston, you could set up a computer containing a database in one location that is accessible by all of the offices via a network. This is not only possible but also safe because databases have a clearly defined structure and also enforce rules that protect the data contained. They also allow more than one person to access the database at the same time and change the data stored; the database management system handles simultaneous changes. Imagine the potential chaos if you used an Excel spreadsheet, and two salespeople change data simultaneously. You want to keep both sets of changes, but whoever saves the spreadsheet last is the person whose changes are stored, overwriting any earlier changes.

Databases also make sharing data between different systems much easier than using proprietary data formats — that is, a format specific to a particular program, manufacturer, or operating system. An Excel spreadsheet, for example, is easily read on a Windows machine with MS Office, but it is more of a challenge to read on a UNIX, Macintosh, or Linux machine because those computers handle data in a different way. Even on a Windows machine, you need to have MS Office installed. You can house a database on a central computer, put the database management system on there, and then enable access via a local network or the Internet.

As an alternative to databases, text files and spreadsheets have one big advantage, which is also their weakness: flexibility. Text files have no real rules. You can insert whatever text data you like wherever you like. To a large extent, spreadsheets are the same. You can ask users to add data in a predefined structure, but you have no real way to enforce such a request. Using databases limits user access to just the data and does not allow users to change the structure.

One final significant advantage of databases is security. Most database management systems allow you to create users in order to specify various levels of security. Before someone accesses the database, he or she must log on as a specific user. Each user has various rights and limits. Someone who maintains the database has full ability to edit data, change the database's structure, add and delete users, and so on. Other users may only have the ability to view data but not change it, or you may even want to limit what data they can view. Many database management systems provide a granular level of security, that is, they are very specific as to what a user can do. They are not just an all-or-nothing approach in which the user either has access or has no access.

Databases are used pretty much everywhere. Data processing played a big part in the development of computers, and even today it is one of their main roles. Nearly every walk of life or business requires a database somewhere along the way. Databases are commonly used on personal computers to store data used locally, and on company networks databases store and share company-wide information. The Internet has seen a big rise in databases used to share information; most online shops of a reasonable size use databases. When you visit online stores of any significant size, a database usually provides all the information on the goods being sold. Rather than every page being created by hand, large merchants use a template for book or CD details, and SQL retrieves the book information from the database. Imagine how much work it'd be if Amazon created every single page by hand!

Databases are great at dealing with large amounts of data that need to be searched, sorted, or regularly updated. As you find out in the next few chapters, databases combined with SQL allow you to get the answers you want in the order you want.

Database Management Systems Used in This Book

Databases are great for storing data, the database management system provides ways of looking at the data, and usually software provided allows you to view the data. But how do you use the data outside of the database management software? The operating system, whether it's Windows, UNIX, Linux, or the Macintosh, provides ways of hooking into the database management system and extracting the data. You need to write programming code to put inside a stand-alone application that the user runs on their computer, or you could set up a Web page to extract data. You're not restricted to certain languages, so long as the language allows you to hook into the database management software.

You can buy any number of different relational database management systems off the shelf, but this book's aim is to present SQL that is standards compliant (more on the standards in the next section) and that works with as wide a range of RDBMSs as possible. However, there are times when the standards don't allow you to do what you want. Other times, you may find that the various DBMS vendors haven't implemented them consistently. This book provides details specific to MS Access, MS SQL Server 2000, IBM DB2, MySQL, and Oracle 10.

Structured Query Language (SQL)

The first questions to ask are what is SQL and how do you use it with databases? SQL has three main roles:

- ☐ Creating a database and defining its structure
- ☐ Querying the database to obtain the data necessary to answer questions
- ☐ Controlling database security

Defining database structure includes creating new database tables and fields, setting up rules for data entry, and so on, which is expressed by a SQL sublanguage called Data Control Language (DCL), covered later in this chapter. The next section discusses querying the database.

Finally, DCL deals with database security. Generally, database security is something that database administrators handle.

Creating SQL every time you want to change the database structure or security sounds like hard work, and it is! Most modern database systems allow you to execute changes via a user-friendly interface without a single line of SQL.

Introducing SQL Queries

SQL queries are the most common use of SQL. A SQL sublanguage called Data Manipulation Language (DML) deals with queries and data manipulation. SQL allows you to pose a query (basically a question)

Chapter 1

to the database, and the database then provides the data that answers your query. For example, with a database that stores details of salespersons, car sales, type of cars sold, and so on, you might want to know how many cars each salesperson sold in each month and how much money they made the company. You could write a SQL query that asks this question and the database goes away and gets the data that answers it. A SQL query consists of various statements, clauses, and conditions. A *statement* is an instruction or a command. For example, “Get me some data” is a statement. A *clause* specifies limits to a statement, the limits being specified using *conditions*. For example, instead of “Get some data,” you might say, “Get data only for the sales that were in the month of May,” where “only for” is the clause that specifies which data to retrieve. The condition is “were in the month of May.” If the data doesn’t meet the condition’s criteria, in this case, “month of May,” then you don’t want it. Written as actual SQL code, this could be something like the following:

```
SELECT CarModel
FROM CarSales
WHERE CarSoldDate BETWEEN 'May 1 2005' AND 'May 31 2005';
```

The `SELECT` statement tells the database system that you want to select some data from the database. You then list the data you want, in this case `CarModel` data, which is a field name. You then specify the place the data needs to be taken from, in this case a table called `CarSales`. Finally, you have a condition. The statement above specifies that you want only the data where certain conditions are true. In this case, the condition is that the `CarSoldDate` is between the first and thirty-first of May 2005. Lots of SQL code like that above is covered in Chapter 3’s discussion of statements, clauses, and conditions.

Comparing SQL to Other Programming Languages

Now that you know what SQL can be used for, you can compare it to other programming languages. To be honest, SQL is quite different from the *procedural* languages such as C++, Visual Basic, Pascal, and other third-generation programming languages, which allow the programmer to write step-by-step instructions telling the computer exactly what to do to achieve a specified goal. Taking the car sales example, your goal might be to select all the information about sales made in July from the New York car showroom. Very roughly, your procedural language might be along the lines of the following:

1. Load the sales data into the computer’s memory.
2. Extract the individual items of data from the sales data.
3. Check to see if each item of data is from the month of July and from the New York showroom.
4. If it is, then make a note of the data.
5. Go to the next item of data and keep going until all items have been checked.
6. Loop through the data results and display each one.

SQL, however, is a *declarative* language, which means that instead of telling it what to do to get the results you want, you simply tell it what you want, and it figures out what to do and comes back with the results. In the car sales example, if you were using SQL, you’d specify the results you want, something like this:

```
SELECT all the data from the sales table WHERE the sales were in July and made at the New York
showroom.
```


The SQL language is actually fairly easy to read. The actual SQL could look like this:

```
SELECT * FROM SalesMade WHERE SaleDate = "July 2005" AND SalesOffice = "New York"
```

The asterisk simply means return the data from all the fields in the record.

You learn a lot more about how the SQL `SELECT` statement works in Chapter 3.

Understanding SQL Standards

As with databases, IBM did a lot of the original SQL work. However, a lot of other vendors took the IBM standard and developed their own versions of it. Having so many differing dialects causes quite a headache for the developer, and in 1986 it was adopted by the standards body the American National Standards Institute (ANSI) and in 1987 by the International Standards Organization (ISO), who created a standard for SQL. Although this has helped minimize differences between the various SQL dialects, there are still differences between them.

The following table gives a brief summary of the various standards and updates to those standards.

Year	Name	Also Known As	Changes
1986	SQL-86	SQL-87 (date when adopted by ISO)	First publication of the ANSI/ISO standard
1989	SQL-89		Only small revision of the original standard
1992	SQL-92	SQL2	Major update of the original standard and still the most widely supported standard
1999	SQL-99	SQL3	Update of the 1992 standard adding new ways of selecting data and new rules on data integrity and introducing object orientation
2003	SQL-2003		Introduced XML support and fields with autogenerated values

This book concentrates on SQL-92, SQL-99, and SQL-2003 because most of their features have been implemented by most relational database management systems (RDBMSs). The SQL you write works on most RDBMSs with only minor modifications. There are times when the various RDBMSs do things so differently that compatible code is impossible without big changes; however, these instances are few and far between in this book.

Although standards are important to help bring some sort of commonality among the various RDBMSs' implementation of SQL, at the end of the day what works in practice is what really counts. Instead of endlessly debating standards, this book provides information to help you in the real world of databases. That said, the next section shows you how to create your own SQL database.

Database Creation

So far, this chapter has examined what a database is and where you might use one. This section takes a more in-depth look at the components of a database, its structure, and the accompanying terminology. Finally, you put the theory into action and set up your own sample database.

Once you grasp the basics, this section discusses how to structure your database in an efficient and easy-to-use manner. Good database design simplifies data extraction and reduces wastage by avoiding data duplication.

By the end of this chapter, you'll have a fully functioning database all ready to go for the next chapter when you use SQL to insert, update, and delete data in a database. Not only that, but you'll have the knowledge to experiment on your own and create your own databases. Before any of that happens, however, you need to know more about organizing and structuring a database.

Organizing a Relational Database

This section examines how database systems are organized and what structures they are made up of. These structures include, among other things, databases, tables, and fields. In database terminology, these structures are called *objects*.

The database management system is the overall program that manages one or more databases. Within each database are the tables, which consist of fields. A field contains a specific item of data about something—for example, the age of a person, their height, their eye color, and so on. A table contains one or more fields, and it's usual for a table to contain information about a specific thing or at least data that is related in some way. For example, data about a person could be stored in the Person table. If the information is about a type of person, for example, employee, you might call your table Employees.

As illustrated by the heading, this section is about relational databases, the key word being *relational*. This concept is explained in more detail shortly, but briefly stated, it means that there is some sort of link between data in one table and data in another table, indicating some sort of relationship. For example, the relationship between car sales and car sales employees could be that a particular salesperson sold a particular car.

Figure 1-2 illustrates the basic structure of a relational database.

At the top of the diagram is the RDBMS. This is the application software that manages the various databases. In Figure 1-2, the RDBMS creates two databases, but a RDBMS can be in control of as few as one database to many, many thousands of databases. The RDBMS is the intelligence behind the database system. The RDBMS does all the work involved in creating and maintaining databases and the structures inside them. Additionally, the RDBMS controls security; performs the actual data insertion, deletion, and retrieval; and allows users to interact with the database system via a management console.

Below the RDBMS itself are the actual databases contained therein. Each database contains a collection of one or more tables. Strictly speaking, your database could contain zero tables, but that would make for a pretty useless database! Databases are self-contained, which means that what happens in one table doesn't affect another table. For example, you can create tables with the same names in each database if

you wish, and the database system doesn't complain. Each database you create receives its own specific name or identifier. How a database system manages databases and tables varies depending on your RDBMS. Microsoft Access, for example, works on only one database at a time, although there are ways of linking one database to another. Each Access database is contained in its own file. Other RDBMSs allow you to manage more than one database from the same console.

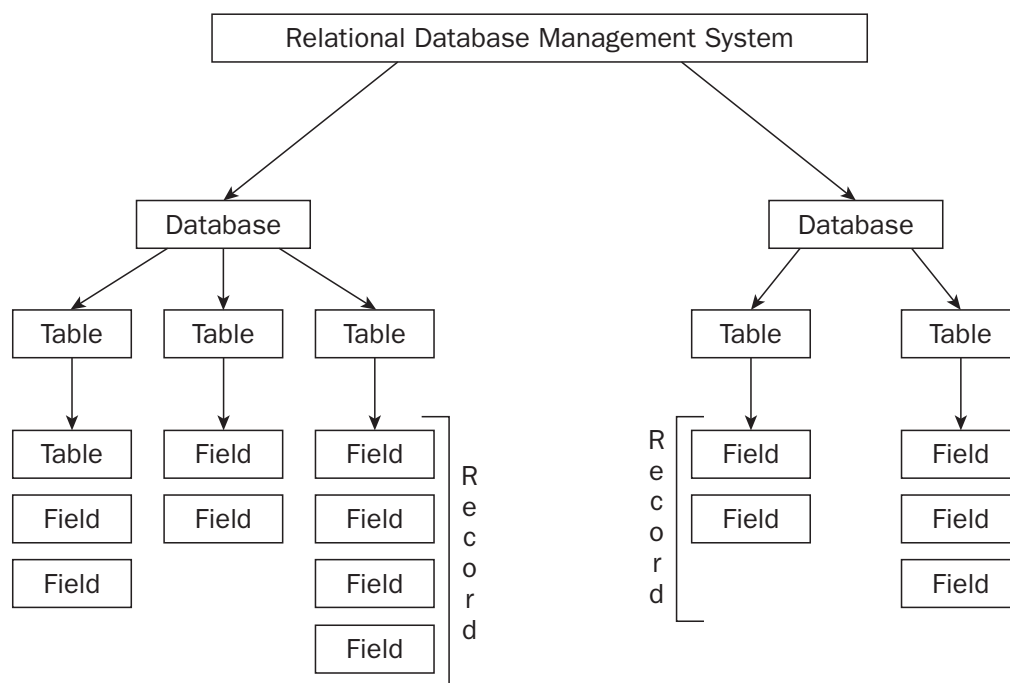


Figure 1-2

Within each database is a collection of tables that contain the records, which hold the data. A good real-world analogy is a train or bus timetable, for example. A simple train timetable could look something like the table shown below:

Start	Destination	Departs	Arrives
London	Manchester	10:15	11:45
Cambridge	Newcastle	9:30	13:55
Lands End	John O'Groats	4:15	23:50
Chester	Liverpool	15:45	16:30
Penzance	Bristol	11:40	18:00

If this were an actual table in your database, you could create a table to hold this data and perhaps call it something stunningly original like `train_times`. The rules as to what you can name tables are fairly flexible but vary a little among RDBMSs. Generally, though, as long as the name doesn't contain punctuation (except things like underscores) and isn't more than 128 characters or so, you are fairly safe.

Chapter 1

From the preceding timetable, you can see that it contains four categories of information: start, destination, time of departure, and time of arrival. In database terminology, these categories are called *fields*, and each field has its own unique name within the table.

Each line in the timetable contains data specific to one aspect of a train schedule. The row contains data pertaining to the train leaving London at 10:15 and arriving in Manchester at 11:45. In a database, the data collectively provided by the fields is called a *record*. The preceding table contains five records. A *column* is all the instances of a particular field from all records in a table. So, in the timetable example, the start column is all the data from the start field for all the records in the table: London, Cambridge, Lands End, Chester, Penzance.

To sum up relational database structure, a RDBMS manages one or more databases, each database contains a collection of one or more tables, and each table contains zero or more records, each record being a collection of fields.

Take what you've learned so far and use SQL to create a database and tables.

SQL Syntax

In programming, *syntax* is the rules to be followed when writing code and the terminology used. Syntax is very much like rules of grammar in languages. For example, the rules of English grammar state that a sentence should end with a period (or full stop as it's known in British English). Of course, there are exceptions to this rule. For example, if you end a sentence with a question, then you use a question mark rather than a period. In SQL there are no sentences; instead there are statements. A statement is a self-contained action. For example, you can use a statement to select certain data, to change the database by adding a new table, and so on. A statement should end with a semicolon; even though many database systems let you get away with leaving it off, it's good practice to include the semicolon.

This book refers to three categories of syntax term: identifiers, literals, and keywords. An *identifier* is something that uniquely identifies something in a database system, using an object such as a database, a table, or field name. If you create a database called `MyDatabase`, then you would say that its identifier is `MyDatabase`. If you create a table called `SalesPeople`, then its identifier is `SalesPeople`. If you need to refer to the `SalesPeople` table, then you use its identifier:

```
SELECT PersonFirstName
FROM SalesPeople;
```

The previous statement selects data from the `SalesPeople` table. The database system knows from which table to retrieve data because you used its identifier, `SalesPeople`.

A *literal* is an actual value, such as 120, Paul, or January 10, 2007. If, for example, you want a list of all salespeople with a first name of Bob, you'd write the following statement:

```
SELECT PersonFirstName, PersonLastName
FROM SalesPeople
WHERE PersonFirstName = 'Bob';
```

This statement uses the literal value of Bob in its comparison. Don't worry if the SQL is still a bit unclear — you take an in-depth look at `SELECT` statements in Chapter 3.

A *keyword* is a word that has some meaning to the database system. For example, if you say, “flob-badob,” people would no doubt wonder what on earth you were talking about! But if you use the word “stop,” it’s a word you know and it has a certain meaning for you. So to the database system, “flob-badob” means nothing, but `SELECT` has a special meaning that the database system acts on. It means, “I want to select some data.” Each keyword has its own rules. If you use `SELECT`, then the database system expects as a minimum a list of data you want to select and where that data is coming from. It also has optional keywords, such as a `WHERE` clause, specifying what sort of results you want. As you meet each new keyword in this book, you also meet what the database system expects as a minimum and what optional parts can be added.

If you’ve come from other programming languages, you might be wondering about code layout. Some languages allow only one statement per line. SQL, however, allows you to spread your statements over one or more lines. For example, both of the following statements are valid:

```
SELECT CarModel FROM Cars WHERE CarMake = 'Ford';

SELECT CarModel;
FROM Cars
WHERE CarMake = 'Ford';
```

Spacing code over more than one line tends to make it more readable, if it’s done logically. The preceding example puts the individual parts of the `SELECT` statement on their own lines.

Well, that’s enough boring syntax for now. You get more detailed syntax discussions on an as-needed basis as you progress through the book. The discussion now turns to creating a database.

Creating a Database

The first step in databases is creating the database. There are two main ways to create a database.

First, many RDBMSs come with a nice, user-friendly front-end interface, which makes the task of creating a new database very easy. In fact, all it takes is a few mouse clicks, entering a name for the database, and away you go. Systems such as MS Access, MS SQL Server, Oracle, and IBM DB2 all provide a front-end interface. MySQL, however, doesn’t come with a default front end, but there are plenty of free ones available, such as MySQL Control Center.

In the case of MS Access, using the program to create a database is the only way to do so. However, other RDBMSs allow you to use SQL to create a database. Each RDBMS has its own way of allowing you to enter and run SQL statements. For example, SQL Server has the Query Analyzer tool, DB2 has the Command Center, and MySQL has the MySQL Control Center (among many other similar tools). Regardless of the tool you choose to use, the SQL required to create a new database is as follows:

```
CREATE DATABASE myFirstDatabase;
```

It really is that easy! Once you become more advanced, you discover a plethora of options you can play with, but for the purposes of this book, the default options used by the `CREATE DATABASE` statement are fine.

The database is called `myFirstDatabase`, but you could have called it pretty much anything. Some restrictions on the name exist, on its length, for example. DB2 limits name length to eight characters, and

Chapter 1

SQL Server limits it to 123 characters. It's safer to stick to letters, numbers, and the underscore character and to avoid any punctuation in the name. For example, `My_db` is fine, but `£$%^my&&&db` is unlikely to work or (and be honest) be easy to pronounce! Numbers are usually fine to include, but most RDBMSs don't allow a database's name to begin with a number. Finally, and it may seem obvious, a database name must be unique within the RDBMS. If you call two databases `myDB`, the RDBMS won't know which one you're referring to when you're writing your SQL code.

What if you want to delete the database? Again, most RDBMSs have a nice and easy user console that allows you to do that, but you can also drop a database using SQL. You don't use *delete database* as you might expect; instead you use the `DROP DATABASE` statement followed by the database's name.

So to drop the `myFirstDatabase` you write the following:

```
DROP DATABASE myFirstDatabase
```

This isn't a command to be used lightly, though! Dropping the database removes it from the RDBMS and you could potentially lose all your data.

Oracle is a bit of an exception when it comes to dropping databases. Instead of the `DROP DATABASE` command, you create the database again! If you already have a database called `myFirstDatabase`, the RDBMS deletes it if you write

```
CREATE DATABASE myFirstDatabase
```

This is something to be very careful of.

After creating a database, the next stage is to add tables to it. However, before you can add tables, you need to look at the concept of data types.

Understanding Data Types

Outside of the world of information technology, you categorize various bits of information into different types quite naturally. You think of the price of goods in a shop as being numerical data. If you ask for directions from New York to Washington, you expect to receive verbal instructions such as "turn left at..." In databases, a *data type* is the classification of different sorts of data being stored, whether the data are numbers, characters, or dates. It helps the database system make sense of the values being inserted into a database. So just as in the world outside databases, you categorize different types of data, but you do so in a more formal way. Returning to the train timetable example, the following table outlines what type of data each field holds:

Field	Data Type	Example
Start	Character data	London, Chester
Destination	Character data	Manchester, Bristol
Departs	Time	10:15, 11:40
Arrives	Time	11:45, 18:00

A perfectly valid question to ask is, “Why have different data types?” Why not just treat everything as text? The main reason is efficiency. The amount of storage space and the speed of access improve when the database knows what sort of data it’s dealing with. For example, the number 243787452 can be stored in as little as 4 bytes of computer memory. Storing the same number as text takes 9 bytes of memory for the character data.

In addition, the data type determines what the RDBMS expects users to do with the data. If you have numerical data, then $123 + 123$ calculates as addition with the answer being 246. If it were text data, the RDBMS would interpret the plus sign as meaning that you want to join the two character strings together to form 123123.

So what are the various data types available? Unfortunately, data type varies among RDBMSs. Added to this conundrum is the problem that while the ANSI SQL standards such as SQL-92, SQL-99, and SQL-2003 define standards for data types, they are far from fully and universally implemented by the various RDBMS manufacturers. However, all is not lost. There’s enough support of the standards for the purposes of this book. Once you have a handle on the basic ANSI SQL data types, researching the data types that your particular RDBMS uses is fairly easy. You can then use them in addition to the data types examined here.

The following table contains a subset of the more commonly used ANSI SQL data types and the name of the same data type in other RDBMSs such as SQL Server, IBM DB2, and so on.

ANSI SQL	MS Access	SQL Server 2000	IBM DB2	MySQL	Oracle 10
Character	char	char	char	char	char
Character varying	varchar	varchar	varchar	varchar	varchar
National character	char	nchar	graphic	char	nchar
National character varying	varchar	nvarchar	vargraphic	varchar	nvarchar
Integer	number (long integer)	int	int	int	int
Smallint	number (integer)	smallint	smallint	smallint	smallint
Real	number (double)	real	real	real	real
Decimal	number (decimal)	decimal	decimal	decimal	decimal
Date	date	datetime	date	date	date
Time	time	datetime	time	time	date

Chapter 1

Although this table includes only a small subset of all the possible data types for all the RDBMSs out there, it's more than enough to get you started. Note that although Oracle does support the `nchar` and `nvarchar` types, it does so only if you create a new database and specify that the character set is a Unicode character set such as `AL16UTF16`. Otherwise, by default it doesn't support `nchar` and `nvarchar`.

The following table describes each data type, roughly how much storage space it uses, and an example of its use. The ANSI names have been used for the data type.

Data Type	Description	Storage Used	Example
character	Stores text data. A character can be any letter, number, or punctuation. You must specify how many characters you want to store in advance. If you actually store fewer than you allow for, the RDBMS adds spaces to the end to pad it out.	One byte per character allocated.	<code>char(8)</code> allocates space for eight characters and takes up approximately 8 bytes of space.
character varying	Similar to character except the length of the text is variable. Only uses up memory for the actual number of characters stored.	One byte per character stored.	<code>nchar(8)</code> allocates space for up to eight characters. However, storing only one character consumes only 1 byte of memory, storing two characters consumes 2 bytes of memory, and so on; up to 8 bytes allocated.
national character	Similar to character, except it uses two bytes for each character stored. This allows for a wider range of characters and is especially useful for storing foreign characters.	Two bytes per character allocated.	<code>nchar(8)</code> allocates space for eight characters and consumes 16 bytes of memory regardless of the number of characters actually stored.
national character varying	Similar to character varying, except it uses 2 bytes to store each character, allowing for a wider range of characters. Especially useful for storing foreign characters.	Two bytes per character stored.	<code>nvarchar(8)</code> allocates space for eight characters. How much storage is used depends on how many characters are actually stored.

Data Type	Description	Storage Used	Example
integer	A whole number between -2,147,483,648 and 2,147,483,647.	Four bytes.	int consumes 4 bytes regardless of the number stored.
smallint	A whole number between -32,768 and 32,767.	Two bytes.	smallint consumes 2 bytes of memory regardless of the number stored.
real	A floating-point number; range is from $-3.40\text{E}+38$ through $3.40\text{E}+38$. It has up to eight digits after its decimal point, for example, 87.12342136.	Four bytes.	real consumes 4 bytes of memory regardless of the number stored.
decimal	A floating-point number. Allows you to specify the maximum number and how many digits after the decimal place. Range is from $-10^{38} + 1$ through $10^{38} - 1$.	5–17 bytes.	decimal(38,12) sets a number that is up to 38 digits long with 12 digits coming after the decimal point.
date	Stores the date.	Four bytes.	date, for example, 1 Dec 2006 or 12/01/2006. Be aware of differences in date formats. In the U.K., for example, "12/01/2006" is actually January 12, 2006, whereas in the U.S. it's December 1, 2006.
time	Stores the time.	Three bytes.	time, for example, 17:54:45.

Note that the storage requirements pertain only to the actual data. The RDBMS usually requires a little bit of extra storage to make a note of where in its memory the data is stored. However, these internal workings are not something you need to worry about unless you're involved in very advanced database work. Also, details may vary depending on the RDBMS system you use.

Although some of the data types are self-explanatory, some of them need a little more explanation. The following sections explain some data types in greater depth, starting with character data types.

Chapter 1

Characters

When you want to store text, you use one of the character data types. Note that the term *string* means one or more characters together. There are four possible variations of character data type selection:

- ☐ Fixed length
- ☐ Variable length
- ☐ Standard 1 byte per character (`char` and `varchar` types)
- ☐ Standard 2 bytes per character (`nchar` or `nvarchar` types)

I'll examine the difference between the fixed- and variable-length data types first. Note the following code:

```
char(127)
```

If you use the preceding code, the RDBMS allocates enough memory to hold 127 characters. If you store only 10 characters, then the other 117 allocated places in memory are filled with spaces, which is fairly wasteful. If you're using only 10 characters, you might wonder whether you can just write

```
char(10)
```

That's fine, but sometimes you may fill all 127 character places.

By contrast, `varchar(127)` doesn't allocate any memory; it simply says to the RDBMS, "I might want to store up to 127 characters but I don't know yet." So if you store only 10 characters, you use only the memory space required for 10 characters. Storing 127 characters is fine, too, though that uses memory required for 127 characters.

At this point, it may seem like a bother to use the fixed character type. Why not always use `varchar`? There are two main reasons. First, inserting and updating fixed character data types is quicker—not by a huge amount, but some databases might be updating tens of thousands of records a second, in which case very small differences can add to one big difference. So where you're storing only a few characters and where speed is of the essence, then the fixed data type wins out.

Second, if you store only a few characters, the memory savings between the two methods is fairly insignificant.

The next variation is between the `char`/`varchar` data types that use just one byte to store a character and the `nchar`/`nvarchar` that use 2 bytes. The 1-byte system originates from the ASCII character set. ASCII (or American Standard Code for Information Interchange) was developed in the early 1960s as a universal way of representing characters on a computer. Computers work only on bits and understand only binary numbers, so characters are stored as numbers, where a number between 0 and 255 represents each letter. For example, the letter A has a numerical value of 65, B has a value of 66, and so on. While this is more than enough to cover letters of the alphabet, numbers, and some punctuation, there are plenty of other characters, especially ones common in foreign languages, that ASCII just can't cope with. To overcome this problem, the Unicode character set was developed. Unicode uses 2 bytes to represent each character and allows you to specify 65,536.

The `char` and `varchar` data types use the one-byte ASCII-based storage. The `nchar` and `nvarchar` support the 2-byte Unicode character set. Which data type you use depends on whether your database requires compatibility with foreign characters. Whichever character set you choose, the usage in SQL code is generally the same, unless the database system has a specific feature designed to work differently based on Unicode or ASCII. The obvious downside of `nchar` and `nvarchar` is that they use twice as much memory to store characters, because they use 2 bytes per character.

Before moving on, you should note something about maximum character storage. MS Access and MySQL can store only a maximum of 255 characters in a character field. To get around this, use the `memo` data type for MS Access, which can store up to 65,535 characters—more than enough for most purposes. For MySQL, if you want to store large amounts of text, use the `text` data type, which also stores a maximum of 65,535 characters. You don't need to specify the maximum number of characters either the `memo` or `text` data type can hold—it's preset by the database system itself.

Numerical Data

The easiest numbers to understand and deal with are the integers: whole numbers rather than numbers with decimal points. They are particularly useful as unique identifiers for a record, something you learn more about later in this chapter when you create the example database and look at primary keys. Fractional numbers are subject to rounding errors and therefore are best avoided if you want a unique identifier for a record. Integer numbers are also less work for the RDBMS, and less work means more speed. Therefore, they are more efficient unless you really do need the fractional part. For example, you'd need the fractional part if you're dealing with money and want to store dollars and cents.

The two integer data types used in this book are `int` and `smallint`. The difference between the two is simply in the size of number they can store and in how many bytes of memory they require to store a number. `smallint` can deal with a range between $-32,768$ and $32,767$, whereas `int` can handle a range between $-2,147,483,648$ and $2,147,483,647$.

The final two numerical data types covered in this chapter can store the fractional parts of numbers: `real` and `decimal`. The `real` data type can store a range of numbers between $-3.40\text{E}+38$ through $3.40\text{E}+38$, though this varies from RDBMS to RDBMS. Note that $3.40\text{E}+38$ is an example of scientific notation. It means the same as 3.4×10 to the power of 38. For example, 539,000,000 could be stated in scientific notation as $5.39\text{E}+8$. This data type is very useful if you have huge numbers but are not too concerned about preciseness. When you store numbers in a `real` data type, you can use either the scientific notation or just the number itself. If the number is too large to store precisely in a `real` data type, the database system converts it to the scientific notation for you, but quite possibly with some loss of accuracy. It's not a data type you'll find yourself using particularly often.

The `decimal` data type is similar to `real` in that it allows you to store floating-point numbers as opposed to whole numbers. A floating-point number is a number with a fractional part—numbers after a decimal point—and that decimal point is not fixed in any particular position (123.445 or 4455.4, for example). Whole numbers, or integers as they are also known, can't store fractional numbers, meaning that they don't store a decimal point. `decimal` is more accurate and flexible than the `real` data type. How can the `decimal` data type be more accurate than the `real` data type, you ask? Surely the answer is either right or wrong. After all, this is math and not sociology, so there's no room for debate. Actually, while the `real` data type can store huge numbers, it does so using scientific notation, so not all the digits are actually stored if the data is over a certain size.

Chapter 1

Look at an example to make this clearer. If you have the number 101236.8375 and try to store it as a `real` data type, the RDBMS stores 101236.84. Why did the 75 at the end disappear? On some RDBMSs, such as SQL Server and DB2, `real` can store only eight digits. The number above is 10 digits long, so the RDBMS rounds off the number and chops off the last 2 digits. Say you have a very large number, such as 101249986.8375. Even after rounding off, the number is still larger than 8 digits, so the RDBMS uses scientific notation and displays 1.0124998E+8.

The `decimal` data type differs from the `real` data type in that it stores all the digits it can. If you store a number larger than it can handle, the RDBMS throws an error saying that there is an overflow. Therefore, the digits on the left of the decimal point are always correct. The `decimal` data type, however, rounds up any digits to the right of the decimal point if it doesn't have enough spare space to display them.

The `decimal` data type's flexibility comes into play when it allows you to specify how many digits you want to store, as well as how many digits can appear on the right side of the decimal point. The following code tells the RDBMS to allocate space for 38 digits, with space reserved for 12 digits after the decimal point:

```
decimal(38,12)
```

This means that the RDBMS stores 101249986.8375 correctly, though it adds zeros after the 8375 to fill in 12 reserved spaces.

The maximum number of digits many RDBMSs allow for the `decimal` data type is 38. The more digits you ask the RDBMS to store, the more storage space you require. So `decimal(9,2)` requires 5 bytes, whereas `decimal(38,2)` requires 17 bytes!

Date and Time

Time is fairly easy to deal with. You simply store the time in the format hours:minutes:seconds. For example, 17:56:22 translates to 5:56 P.M. and 22 seconds.

Most RDBMSs use the 24-hour clock, so to store 5:36 P.M., you write 17:36:00.

Some RDBMSs don't have a separate date and time, but instead combine them into one, in which case the date goes first and then the time in the format just mentioned of hh:mm:ss. For example, you may encounter a date and time similar to the following: 1 Mar 2006 10:45:55.

Whereas the format for time is fairly standard, dates can have many possible variations. For example, all of the following are valid in either the United States or Europe:

- ☐ 12 Mar 2006
- ☐ Mar 12, 2006
- ☐ 12 March 2006
- ☐ 12/03/2006
- ☐ 03/12/2006
- ☐ 03-12-2006

Most RDBMSs handle the most common variations, like the preceding examples. However, the biggest problem arises when you specify the month by number instead of name — 03/12/2006, for example. An American would read this as the 12th day of March 2006, and a resident of the United Kingdom would see the date as the 3rd day of December 2006 — quite a difference! There is little else in database development that produces bigger headaches than dates!

Even worse is when the computer the RDBMS is running is set to the American format and the computer accessing the data from the database is set to the U.K. format. And with many companies having U.S. and foreign-based offices, this can often happen.

Whenever possible, avoid the number format, such as 12/07/2006, and instead use the month's name or at least the abbreviation of its name (12 Jul 2006, for example). Unfortunately many RDBMSs return the date format as 12/07/2006 even if you stored it as 12 Jul 2006. In this case, use formatting commands, which are covered in Chapter 5, to circumvent this problem. Also try to make sure you know the format that the RDBMS server is using.

Now that the brief introduction to data types is over with, you can move on to something a bit more interesting and hands-on — creating tables!

Creating, Altering, and Deleting Tables

This section discusses the basics of creating a table using SQL. It shows you how to create a new table, how to modify existing tables, and finally, how to delete tables that you no longer need. After that, using what you've learned, you create the tables for the book's example database.

Creating a Table

To create a table, use SQL's `CREATE TABLE` statement. Creating a basic table involves naming the table and defining its columns and each column's data type.

More advanced table options and constraints are covered in Chapter 4.

The following is the basic syntax for creating a table:

```
CREATE TABLE name_of_table
(
    name_of_column column_datatype
)
```

`CREATE TABLE` is the keyword telling the database system what you want to do — in this case, you want to create a new table. The unique name or identifier for the table follows the `CREATE TABLE` statement. Then in brackets comes the list defining each column in the table and what sort of data type it is. The syntax becomes clearer with an example.

Chapter 1

The following SQL creates a table based on the earlier train timetable example:

```
CREATE TABLE Train_Times
(
    start_location varchar(75),
    destination varchar(75),
    departs time,
    arrives time
);
```

MS SQL Server doesn't have the time data type, so you need to change the data type to datetime. For Oracle, you need to change the data type to date rather than time, because Oracle's date data type stores both date and time.

For this and all of the examples in this book, load up the tool that came with your RDBMS that allows SQL code to be written and run. You can find installation details in Appendix B at the end of the book.

Examining the code line by line, first you specify that you want to create a table called `Train_Times`:

```
CREATE TABLE Train_Times
```

Then, inside brackets, you specify the four fields that make up each record. For each record, you need to identify the field name and data type:

```
(
    start_location varchar(75),
    destination varchar(75),
    departs time,
    arrives time
)
```

A comma must separate each field definition. Notice the SQL's neat layout, with the `CREATE TABLE` statement and each field definition on separate lines. Such layout is not compulsory. In fact, you could cram the whole lot on one line. However, laying out the code on separate lines makes the code a lot more readable and therefore easier to write and makes it easier for you to fix errors, or *debug*, if things go wrong.

You can see that creating tables is pretty easy. There are more complexities with table creation than listed here, and these are examined in Chapter 4. The next issue is how to alter tables. Say you want to add a new field, delete an existing one, or perform other routine table maintenance. The good news is that SQL allows you to perform all of these functions with one statement: `ALTER TABLE`.

Altering an Existing Table

The key to changing an existing table is the `ALTER TABLE` statement. This one statement allows you to add and delete columns in an existing table. What the ANSI SQL standard `ALTER TABLE` statement doesn't let you do, however, are things like changing the data type of an existing column. However, many RDBMSs have extended the `ALTER TABLE` statement and include their own way of changing column definitions.

To add a new column, use the basic syntax shown below:

```
ALTER TABLE name_of_table
  ADD name_of_field data_type
```

`ALTER TABLE` is the keyword that tells the database system what to do. After the `ALTER TABLE` statement, you supply the name of the table being altered. Finally, the syntax above tells the database system that you want to add a new column and then supplies the name of the column and its data type—in much the same way that you define column name and data type when creating a table.

To delete an existing column, the syntax is identical except you now tell the database system that you want to delete a column and supply that column's name:

```
ALTER TABLE name_of_table
  DROP COLUMN name_of_field
```

A couple of examples make this a bit clearer. In order to add a column called `runs_at_weekend` with the data type `char(1)` to the `Train_Times` table, use the following SQL:

```
ALTER TABLE Train_Times
  ADD runs_at_weekend char(1);
```

To delete the same column, you write the following:

```
ALTER TABLE Train_Times
  DROP COLUMN runs_at_weekend;
```

IBM DB2 doesn't support the `DROP COLUMN` statement.

Remember, as with dropping a table, dropping a column most likely permanently deletes the data in that column. Use the `DROP COLUMN` statement carefully!

Finally, the next section discusses how to delete an existing table.

Deleting an Existing Table

By now you're probably seeing a pattern emerge when it comes to deleting things, so yes, you guessed it, deleting a table involves using the `DROP TABLE` statement. The basic syntax is

```
DROP TABLE name_of_table
```

To delete the `Train_Times` table, you write

```
DROP TABLE Train_Times
```

This section only scratches the surface of adding and altering tables and columns. Chapter 4 identifies potential complications that may arise when dropping a table that contains data that another table relies upon.

Chapter 1

You should now know enough fundamentals to begin creating usable databases and tables. The final section of this chapter walks you through creating the example database that you use throughout the book. First, though, you need to know how to use good database design techniques to create an effective database.

Good Database Design

This section examines some basic rules and ideas that help you develop an effective and well-designed database. While Chapter 4 takes a more in-depth look, this chapter provides enough of the fundamentals to get you started. Begin with the all-important first step: consider why you even need a database.

Obtaining and Analyzing Your Data Needs

Before you create the database and write a single SQL statement, you need to sit down and think about why you're creating the database in the first place. This doesn't only mean because someone just paid you huge piles of cash to do so, though that is a nice benefit! It means to ask yourself, or whomever the database is for, what sort of data will be stored and what sort of answers the data needs to provide. For example, imagine that you decide to set up a film club, and being a high-tech sort of person, you decide that keeping membership details on the back of an old envelope is just too low-tech! So you decide that you need a database to help run the club. First of all, you need to sit down and think about why you need the database and what data you need to store. The club membership database may need to store details of club members. You might also like to know how popular the club is by keeping details of meetings and attendance. A good start would be to write a list of all the information you want to store.

Say you want to be able to contact each member to send out information by post and email. You also want to send them a birthday card each birthday (how nice are you!). Finally, you want to make sure that members pay their annual memberships fees, so you need to keep track of when they joined. The following list summarizes what you want to store:

- ☐ Full name
- ☐ Date of birth
- ☐ Address
- ☐ Email address
- ☐ Date member joined club

The aim with meetings is to keep track of how popular meetings are at various locations and who attends regularly. The following is the sort of data to store in order to keep track of such information:

- ☐ Meeting date
- ☐ Location
- ☐ Who attended the meeting

Now that you know what you want to store, the next step is to divide the data logically to get some idea of your table structure.

Dividing Data Logically

For now, don't worry about table names, column names, or data types; rather, just get a rough idea of what table structure you want to use.

In a first attempt, you might decide to just lump the whole lot into one huge table, including the following information:

- ☐ Full name
- ☐ Date of birth
- ☐ Address
- ☐ Email address
- ☐ Date member joined club
- ☐ Meeting date
- ☐ Location
- ☐ Whether member attended meeting

If you were to use the preceding example data, your records could look like the following table:

Name	Date of Birth	Address	Email	Date of Joining	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

Seems like a reasonable start. However, you do have one problem: How do you store details of more than one meeting? One option would be to simply create a new record for each meeting, something akin to the following table:

Chapter 1

Name	Date of Birth	Address	Email	Date of Joining	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

Although that method seems to work fine, it's bad news in terms of efficiency and becomes unmanageable later on as more members join and more meetings are held.

What's so wrong with it?

First, you have unnecessary data duplication. Each time you hold a new meeting, you need to store not only the meeting details, but also, because they are in the same record, the members' details again. Your database would become huge in size quite quickly. Additionally, a lot of work is involved in actually updating the data. If Jane, for example, moves into a new house, then you need to update not just one record, but every single record relating to her. Another problem arises when you try to retrieve member details. Choosing which of the members' records to retrieve the data from would be extremely difficult; after all, you can choose from more than one record.

Rethinking your organization, you need to store details of more than one meeting, so instead of having more than one record per person, instead create new columns for each meeting so that your fields include the following:

- ☐ Full name
- ☐ Date of birth
- ☐ Address
- ☐ Email address
- ☐ Date member joined club
- ☐ Meeting date 1
- ☐ Location 1
- ☐ Whether member attended meeting 1

- ☐ Meeting date 2
- ☐ Location 2
- ☐ Whether member attended meeting 2
- ☐ Meeting date 3
- ☐ Location 3
- ☐ Whether member attended meeting 3

Organizing your columns like this saves some data duplication but results in inflexibility. Say you want to keep records of the last 10 meetings. To do so, you need a total of 30 columns in the table, and you would have to redesign the database every time you want to record more meetings. Such organization also makes writing SQL statements to extract the data harder and more long-winded.

What you need to do is split apart the data into logical parts. In this case, you are collecting data about two distinct things: club members and meeting attendance. An obvious relationship exists between the two. After all, without any members, there is no point in having meetings! That said, split your data into club member details in one table and meeting attendance data in a second table.

In the MemberDetails table, include the following:

- ☐ Full name
- ☐ Date of birth
- ☐ Address
- ☐ Email address
- ☐ Date member joined club

In the meeting attendance details table, include the following:

- ☐ Full name
- ☐ Date of the meeting
- ☐ Location
- ☐ Whether member attended meeting

Now the MemberDetails table might look something like this:

Name	Date of Birth	Address	Email	Date of Joining
Martin	Feb 27, 1972	1 The Avenue, NY	martin@some.com	Jan 10, 2005
Jane	Dec 12, 1967	33 Some Road, Washington	Jane@server.net	Jan 12, 2005
Kim	May 22, 1980	19 The Road, New Townsville	kim@mail.com	Jan 23, 2005

Chapter 1

The Attendance table could look like this:

Name	Meeting Date	Location	Did Member Attend?
Martin	Mar 30, 2005	Lower West Side, NY	Y
Martin	April 28, 2005	Lower North Side, NY	Y
Jane	Mar 30, 2005	Lower West Side, NY	N
Jane	April 28, 2005	Upper North Side, NY	Y
Kim	Mar 30, 2005	Lower West Side, NY	Y
Kim	April 28, 2005	Upper North Side, NY	Y

Splitting member details and meeting details into two tables saves a lot of data redundancy. Member details are stored only once, and the only redundant data is the name, which links the two tables together. This is a reasonable start to database table design, so now turn your attention to defining the data types.

Selecting Correct Data Types

After getting a rough idea of the table design, the next step is to choose the data types for each field. Sometimes this is fairly obvious. For example, storing a person's name in a numerical field doesn't make sense!

However, there are times when perhaps the data type is less obvious. For example, although a telephone number is a number, you store it in a character field for a couple of reasons. First, telephone numbers are rarely involved in mathematical calculations. Second, sometimes telephone numbers start with zeros. For example, 077123333 would be stored as 77123333 in a numerical field; the RDBMS removes the leading zero because it's not important to the number's value as a number, though the zero is important to the number's value as a telephone number.

Consider these factors when choosing a data type to use:

- ❑ **The data's use:** Is the data intended for mathematical calculations? Does the data represent date or time situations? Does the data simply display text-based information?
- ❑ **The data's size:** Choose a data type that covers the largest value you reasonably expect to store. For example, if you expect people with names 100 characters long, then ensure that the character field could handle it. If the data is a number, ensure that the largest number fits into the field.
- ❑ **Correct information storage:** For example, if you use an integer data type to store a monetary value, then you lose any fractional parts. So, the integer data type stores \$2.99 as 2. Even with data types that store the fractional parts, you may find that the database rounds them up or down, resulting in incorrect values—especially with the real data type. Use a field specific to money if your RDBMS supports it, or at least use `DECIMAL(10, 2)`.
- ❑ **Non-English characters:** If you need to store non-English characters, choose one of the `nchar` or `nvarchar` data types for your text data.

Generally speaking, picking a data type is common sense. Using the Film Club database's MemberDetails table, pick some names and data types:

Field Name	Data Type
Name	<code>varchar(75)</code>
DateOfBirth	<code>date</code>
Address	<code>varchar(200)</code>
Email	<code>varchar(200)</code>
DateOfJoining	<code>date</code>

As you can see, it's mostly common sense when choosing data types. A name, a street address, or an email address are text-based information, so you would choose the `text` data type. Likewise, date of birth, a date, is stored in the `date` data type. The size of the fields, however, contains an element of guesswork. You can't be 100% sure that there isn't someone out there with a name longer than 75 characters, but a good guess is that they are few and far between. To be on the safe side, go for an estimate that holds more characters than you ever expect to be stored.

The data types for the Attendance table are the following:

Field Name	Data Type
Name	<code>varchar(75)</code>
MeetingDate	<code>date</code>
Location	<code>varchar(200)</code>
MemberAttended	<code>char(1)</code>

Again it's fairly easy to choose, with Name and Location being text-based data — so a character data type, here `varchar`, is the best choice. MeetingDate is a date field, so the `date` data type has been chosen. MemberAttended is unusual. You just want to store whether a member did or did not attend. To do this, you can go for a yes or no choice, with *yes* represented by a Y, and *no* represented by an N. A character data type is best, and because you only ever plan to store one character, it's more efficient to use the fixed `char` data type and specify that the field can hold just one character.

Using a Primary Key

A *primary key* is a field or fields that uniquely identify a record from the other records in the database table. Continuing with the film club example, you might consider yourself the first club member, so keeping a database record of membership is pretty easy. As a few more people join, you decide to create a database and store their details as well, including name, age, address, and so on. When there are just a few people in the club, and, hence, just a few records in the database, using members' names to identify people in the database is probably going to work fine. Imagine, though, that the club expands massively

Chapter 1

to thousands of members. Suddenly the risk of there being two, three, or more identical names gets ever higher, and your ability to select the right person from the database gets lower! In such cases, you need some sort of unique identification. You could use name and age in combination, but the problem is that people age, so the unique identifier changes, which makes keeping track of people quite hard. There's also the small risk of two people with the same name and age. You could use name and address, but again, addresses change, and also addresses are quite long, which slows down the database in terms of searching and sorting data.

You may have already guessed that the answer to the problem is to give everyone a unique identifier that only they have. In a club, you might call the unique identifier something like the `MemberId`. And this is all a primary key is — a unique identifier. You know that `MemberId 1234432` applies to just one person, and if you select that data from the database, you are guaranteed to get the record you want.

Primary keys also link one table to another. For example, if you have one table with member details in it and a second table that contains details of meeting attendance, you could link the two tables together with a primary key. For example, the `MemberDetails` table could look like this:

Field Name	Data Type
<code>MemberId</code>	<code>integer</code>
<code>Name</code>	<code>varchar(75)</code>
<code>DateOfBirth</code>	<code>date</code>
<code>Address</code>	<code>varchar(200)</code>
<code>Email</code>	<code>varchar(200)</code>
<code>DateOfJoining</code>	<code>date</code>

Note that the `MemberId` column is the primary key column for this table.

Your Attendance table could look like this:

Field Name	Data Type
<code>MeetingDate</code>	<code>date</code>
<code>Location</code>	<code>varchar(200)</code>
<code>MemberAttended</code>	<code>char(1)</code>
<code>MemberId</code>	<code>integer</code>

Note that the `MemberId` column is the *foreign key* column for this table. A foreign key contains the value of a primary key in another table, allowing the second table to reference the first. In the film club

example, the `MemberId` field links the member in the `Attendance` table with their full details in the `MemberDetails` table.

You have a primary key for the `MemberDetails` table, `MemberId`, but what about for the `Attendance` table? The `Attendance` table has a unique identifier — the combination of `MeetingDate` and `MemberId`, because it's not possible for a member to attend the same meeting twice! Therefore, using those two columns in combination to provide unique identification is safe. This provides a nice segue to the second point about primary keys: They don't have to be composed of just one column, as in the case of the `Attendance` table. If you wanted to, you could create a unique meeting ID, but in this situation a unique meeting ID is not necessary and wastes storage space. If speed of data retrieval were a vital issue, you might consider using a primary key, but most of the time it's not necessary.

A primary key simply involves having a column to store a unique value and then generating that value. However, most RDBMSs also allow you to specify a column as a primary key, and in doing so, the RDBMS manages data entry. In particular, you can apply constraints to exactly what data can go in the field. For example, constraints prevent you from having two records with the same primary key value. After all, the point of a primary key is that it's unique. Chapter 4 delves deeper into the more advanced and tricky aspects of primary keys and constraints, but now you have the requisite knowledge to create your first full-fledged database. The database you create in the next section is the one you use throughout the book.

Creating the Example Database

The example database continues the previous discussion of the Film Club database. You need to create a database to store the tables. You can call your database something like `Film Club`, though for the purposes of this book, the database's name is not important. Appendix B has all the specifics needed for creating a blank example database in either Access, SQL Server, DB2, MySQL, or Oracle. Once you've created the `Film Club` database, it's time to start populating it with tables.

The basic premise is that you're running a film club, and you want a database that stores the following information:

- ☐ Club member details, such as name, address, date of birth, date they joined, and email address
- ☐ Meeting attendance details
- ☐ Film details
- ☐ Members' film category preferences

These are the requirements now, though later in the book, you develop and expand the database.

You've already established the club membership's details, but notice the additional fields in the following table:

Chapter 1

Field Name	Data Type	Notes
MemberId	integer	Primary key.
FirstName	nvarchar(50)	Change data type to vargraphic(50) in IBM DB2 and to varchar(50) in MySQL and MS Access. In Oracle nvarchar is not available with the default character set; change to varchar. You must have selected Unicode character set when creating the database to use nvarchar.
LastName	nvarchar(50)	Change data type to vargraphic(50) in IBM DB2 and to varchar(50) in MySQL and MS Access. In Oracle nvarchar is not available with the default character set; change to varchar. You must have selected Unicode character set when creating the database to use nvarchar.
DateOfBirth	date	Change data type to datetime in MS SQL Server.
Street	varchar(100)	
City	varchar(75)	
State	varchar(75)	
ZipCode	varchar(12)	
Email	varchar(200)	
DateOfJoining	date	Change data type to datetime in MS SQL Server.

Notice that the name and address fields are split into smaller parts. Name is split into FirstName and LastName; address is split into Street, City, State, and ZipCode. Splitting name and address data makes searching for specific data more efficient. If you want to search for all members in New York City, then you simply search the City field. If you store street, city, state, and zip code in one address field, searching by city is very difficult.

Create the SQL table creation code. If you prefer, you can use your RDBMS's management console to create the table.

```
CREATE TABLE MemberDetails
(
    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
);
```

Depending on which RDBMS you're using, you need to change some of the data types as outlined in the preceding table. Also, if you're using IBM's DB2 then varchar type is vargraphic. If you're using MS SQL then date is datetime. In MYSQL and MS Access nvarchar needs to be varchar.

The next table to create is the Attendance table, which contains the following fields and data types:

Field Name	Data Type	Notes
MeetingDate	date	Change data type to datetime if using MS SQL Server
Location	varchar(200)	
MemberAttended	char(1)	
MemberId	integer	Foreign key linking to MemberDetails table

The Attendance table is unchanged from the previous discussion of the Film Club database. The SQL to create the Attendance table is as follows:

```
CREATE TABLE Attendance
(
    MeetingDate date,
    Location varchar(200),
    MemberAttended char(1),
    MemberId integer
);
```

What could the unique primary key be for this table? If you assume that you have only one meeting a day in any one location, then a combination of the meeting date and meeting location provides a unique reference. If it were possible, there could be one or more meetings on the same day in the same location, but at different times. In such an instance, you need to store meeting time as well as the meeting date — by having an extra column called MeetingTime. You could then use MeetingDate, MeetingTime, and Location to provide a unique key, or you could create a column called MeetingId containing a unique integer for each meeting to use as the primary key. This sort of situation requires you to go back and ask whomever the database is for what their requirements are and then design your database based on the information they provide, because it's better to get it right from the start.

You still have to add tables to store the following information:

- ☐ Film details
- ☐ Members' film category preferences

A new table called Films needs to be created. The information to be stored in the Films table is as follows:

- ☐ Film name
- ☐ Year of release
- ☐ Brief plot summary
- ☐ Film's availability on DVD
- ☐ User film ratings (for example, 1–5, where 1 is awful beyond belief and 5 is an all-time classic)

Chapter 1

Finally, you need to assign each film a category — for example, horror, action, romance, and so on.

The following table outlines the Films table's contents:

Field Name	Data Type	Notes
FilmId	integer	Primary key
FilmName	varchar(100)	
YearReleased	integer	
PlotSummary	varchar(2000)	Change to memo data type if using MS Access or to the text data type if using MySQL
AvailableOnDVD	char(1)	
Rating	integer	
CategoryId	integer	Foreign key

Before going further, there are two points to note regarding the field definitions. First, the PlotSummary field is a varchar data type, except when used in MS Access, which only supports varchar with a size of up to 255 characters. Because you need to store up to 2000 characters, you must use Access's memo field, which allows you to store up to 65,536. You can't specify length; it's set at a maximum of 65,536 characters, which is equivalent to a varchar(65536) data type.

The second point to note pertains to the CategoryId field. It is a foreign key field, which means that its value is the primary key field in another table, and it provides a way of relating the two tables. The table containing the primary key to which the Films table links is the FilmCategory table, which you will create shortly. First, though, the SQL to create the Films table is as follows:

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

After creating the Films table, you can create the FilmCategory table, which contains the following data:

Field Name	Data Type	Notes
CategoryId	integer	Primary key
Category	varchar(100)	

The FilmCategory table is very small and simple. The SQL you need to create it is likewise very simple:

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

The final table you need to create is called FavCategory, and it stores each member's favorite film categories:

Field Name	Data Type	Notes
CategoryId	integer	Foreign key
MemberId	integer	Foreign key

As you can see, it's a very simple table! Both CategoryId and MemberId are foreign keys, the first from the Category table and the second from the MemberDetails table. Combined, they make up a unique primary key. The SQL to create the FavCategory table is as follows:

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

Creating the FavCategory table completes the basic database structure. As you can see, creating a database is actually fairly easy! Chapter 4 covers more complex examples and options, but what you learned in this chapter lays a good foundation of database creation.

Summary

This chapter walked you through the fundamentals of SQL and database design and provided you with instruction on how to write the SQL code necessary to create a database's structure. You now have enough knowledge to start designing your own databases.

In this chapter, you discovered the following:

- ❑ Databases are an efficient way to store large amounts of raw data. They don't process the data; that's left to the application that uses the data.
- ❑ Databases make sharing data easier than do other means, such as text files, spreadsheets, or other documents. They also allow secure data sharing and allow you to define the levels of user access. You can limit what you let others do to your database's data.
- ❑ Relational databases contain tables and fields and provide ways of relating data in different tables and ways of ensuring that any data entered is valid and doesn't corrupt the database.

Chapter 1

- ❑ Databases are part of a larger software application called a database management system (DBMS).
- ❑ SQL is a declarative programming language, that is, you use it to specify the answers you want and leave the DBMS to work out how to get them.

After getting the basics under your belt, you got down to some practical work. In particular, you created a database and learned the basics of SQL and table creation. In doing so, you found out the following:

- ❑ How databases are organized. You saw that databases are composed of tables, which themselves are composed of records, and that each record is split into fields or columns.
- ❑ How to create a database. You learned that you can create a database either with the RDBMS's management tools or with SQL statements.
- ❑ Different types of data are stored differently. You learned that databases support a number of different data types for storing text (`char` and `varchar`), numbers (`integer`, `real`, and `decimal`), and time and dates (`time` and `date`). These are just the basic data types, and most RDBMSs support many more data types.
- ❑ The principles of good database design.

Finally, at the end of the chapter, you created the book's example database using the techniques and code covered earlier in the chapter. In the next chapter, you learn how to add, update, and delete data using SQL. Don't forget to have a go at the exercises!

Exercises

1. As it happens, the film club holds meetings regularly in several different locations, which means a lot of redundancy in the Attendance table. What changes could you make to the database table structure?
2. Write the necessary SQL to complete the changes required by Exercise 1 and at the same time split the location's address details into street, city, and state.