Chapter 26

Relational Databases

Chapter Goals

- To understand how relational databases store information
- To learn how to query a database with the Structured Query Language (SQL)
- To connect to a database with an application programming interface (API)
- To write database programs that insert and query data in a relational database

When you store data, you want to be able to add more data items, remove data, change data items, and find items that match certain criteria. However, if you have a lot of data, it can be difficult to carry out these operations quickly and efficiently. Because data storage is such a common task, special *database management systems* (DBMS) have been invented that allow the programmer to think in terms of the data rather than file storage. In this chapter you will learn how to use SQL, the Structured Query Language, to query and update information in a relational database, and how to access database information from C++ programs.

Chapter Contents

26.1 Organizing Database Information

26.1.1 Database Tables

A relational database stores information in tables. Each table column has a name and a data type.

A relational database stores information in *tables*. Figure 1shows a typical table. As you can see, each *row* in this table corresponds to a product. Each row contains a sequence of *fields*, with one field per column. The *column headers* correspond to attributes of the product: the product code, description, and unit price. Note that all items in a particular column have the same type: product codes and description are strings, unit prices are floating-point numbers. The allowable column types differ somewhat from one database to another. Table 1 shows types that are commonly available in relational databases that follow the SQL (Structured Query Language, often pronounced "sequel") standard. (See [1] for more information on the SQL standard.)

Product

Product_Code	Description	Price
116-064	Toaster	24.95
257-535	Hair dryer	29.95
643-119	Car vacuum	19.99

Figure 1 A Product Table in a Relational Database

Table 1 Some Standard SQL Types and Their Corresponding C++ Types

SQL Data Type	C++ Data Type
`INTEGER or INT`	`int`
`REAL`	`float`
`DOUBLE`	`double`
`DECIMAL(n, x)`	Fixed-point decimal numbers with *n* total digits and *x* digits after the decimal point.
`BOOLEAN`	`bool`
`CHARACTER(x) or CHAR(x)`	Fixed-length string of length *x*. Similar to `string`.

SQL (Structured Query Language) is a command language for interacting with a database.

Most relational databases follow the SQL standard. SQL commands are used to interact with the database. There is no relationship between SQL and C++—they are different languages. However, as you will see later in this chapter, you can use C++ to send SQL commands to a database.

For example, here is the SQL command to create a product table:

```
CREATE TABLE Products
(
   Product_Code CHAR(11),
   Description CHAR(40),
   Unit_Price DECIMAL(10, 2)
)
```

Use the SQL commands `CREATE TABLE` and `INSERT INTO` to add data to a database.

Unlike C++, SQL is not case-sensitive. For example, you could have spelled the command `create table` instead of `CREATE TABLE`. However, as a matter of convention, we will use uppercase letters for SQL keywords and mixed case for table and column names.

To insert rows into the table, use the `INSERT` command. Issue one command for each row, such as

```
INSERT INTO Products
   VALUES ('257-535', 'Hair dryer', 29.95)
```

As you can see, SQL uses single quotes (`'`), not double quotes, to delimit strings. What if you have a string that contains a single quote? Rather than using an escape sequence (such as `\'`) as in C++, you write the single quote twice, such as

```
'Sam''s Small Appliances'
```

If you create a table and subsequently want to delete it, use the `DROP TABLE` command. For example,

```
DROP TABLE Test
```

PRODUCTIVITY HINT 26.1

Stick with the Standard

The C++ language is highly standardized. You will rarely find compilers that allow you to specify C++ code that differs from the standard. However, SQL implementations are often much more forgiving. For example, many SQL database systems allow you to use a C++-style escape sequence such as

```
'Sam\'s Small Appliances'
```

in a SQL string. Probably the vendor thought that this would be "helpful" to programmers who are familiar with C++.

Unfortunately, this is an illusion. If you deviate from the standard, you limit portability. Suppose you later want to move your database code to another database system, perhaps to improve performance or to lower the cost of the database software. If the other database system hasn't implemented a particular deviation, then your code will no longer work and you need to spend time fixing it.

To avoid these problems, you should stick with the standard. With SQL, you cannot rely on your database to flag all errors—some of them may be considered "helpful" extensions. That means that you need to know the standard and have the discipline to follow it.

26.1.2 Linking Tables

If you have objects whose data fields are strings, numbers, dates, or other types that are permissible as table column types, then you can easily store them as rows in a database table. For example, consider the following `Customer` class:

```
class Customer
{
   ...
private:
   string name;
   string address;
   string city;
   string state;
   string zip;
};
```

It is simple to come up with a table structure for storing customers—see Figure 2.

For other objects, it is not so easy. Consider an invoice. Each invoice object contains a pointer to a customer object.

```
class Invoice
{
   ...
private:
   Customer* cust;
   ...
};
```

Customer

Name	Address	City	State	Zip
CHAR (40)	CHAR (40)	CHAR (30)	CHAR (2)	CHAR (10)
Sam's Small Appliances	100 Main Street	Anytown	CA	98765

Figure 2 A Customer Table

You cannot simply come up with a single column holding a customer, because `Customer*` isn't a SQL type. Table entries are very different from C++ variables. The types of table entries are restricted to SQL types.

Of course, you might consider simply entering all the customer data into the invoice table—see Figure 3. However, this is actually not a good idea. If you look at the sample data in Figure 3, you will notice that Sam's Small Appliances has two invoices, numbers 11731 and 11733. All information for the customer was *replicated* in two rows.

This replication has two problems. First, it is wasteful to store the information multiple times. If the customer places many orders, then the replicated information can take up a lot of space. More importantly, the replication is *dangerous*. Suppose the customer moves to a new address. Then it would be an easy mistake to update the customer information in some of the invoice records and leave the old address in place in others.

In a C++ program, you don't have either of these problems. Multiple `Invoice` objects can contain pointers to a single shared `Customer` object.

You should avoid rows with replicated data. Instead, distribute the data over multiple tables.

The first step in achieving the same effect in a database is to organize your data into multiple tables as in Figure 4. Dividing the columns into two tables solves the replication problem. The customer data are no longer replicated—the Invoice table contains no customer

Invoice

Invoice_ Number	Customer_ Name	Customer_ Address	Customer_ City	Customer_ State	Customer_ Zip	...
INTEGER	CHAR (40)	CHAR (40)	CHAR (30)	CHAR (2)	CHAR (10)	...
11731	Sam's Small Appliances	100 Main Street	Anytown	CA	98765	...
11732	Electronics Unlimited	1175 Liberty Ave	Pleasantville	MI	45066	...
11733	Sam's Small Appliances	100 Main Street	Anytown	CA	98765	...

Figure 3 A Poor Design for an Invoice Table with Replicated Customer Data

Invoice

Invoice_ Number	Customer_ Number	Payment
INTEGER	INTEGER	DECIMAL (10, 2)
11731	3175	0
11732	3176	249.95
11733	3175	0

Customer

Customer_ Number	Name	Address	City	State	Zip
INTEGER	CHAR (40)	CHAR (40)	CHAR (30)	CHAR (2)	CHAR (10)
3175	Sam's Small Appliances	100 Main Street	Anytown	CA	98765
3176	Electronics Unlimited	1175 Liberty Ave	Pleasantville	MI	45066

Figure 4 Two Tables for Invoice and Customer Data

information, and the Customer table contains a single record (row) for each customer. But how can we refer to the customer to which an invoice is issued? Notice in Figure 4 that there is now a `Customer_Number` column in *both* the Customer table and the Invoice table. Now all invoices for Sam's Small Appliances share only the customer number. The two tables are linked by the `Customer_Number` field. To find out more details about this customer, you need to use the customer number to look up the customer in the customer table.

Note that the customer number is a *unique identifier*. We introduced the customer number because the customer name by itself may not be unique. For example, there may be multiple Electronics Unlimited stores in various locations. Thus, the customer name alone does not uniquely identify a record, so we cannot use the name as a link between the two tables.

A primary key is a column (or set of columns) whose value uniquely specifies a table record.

In database terminology, a field (or combination of fields) that uniquely identifies a row in a table is called a *primary key*. In our Customer table, the `Customer_Number` column contains a primary key. Not all database tables need a primary key. A primary key is needed only if you want to establish a link to another table. For example, the Customer table needs a primary key so you can link customers to invoices.

A foreign key is a reference to a primary key in a linked table.

When a primary key is linked to another table, the matching field (or combination of fields) in that table is called a *foreign key*. For example, the `Customer_Number` in the Invoice table contains a foreign key, linked to the primary key in the Customer table. Unlike primary

keys, foreign keys need not be unique. For example, in our Invoice table we have several records that all have the same value for the `Customer_Number` foreign key.

Note that the linkage between tables is not automatically maintained by the database. A change to a `Customer_Number` in the Customer table will not automatically change that number in the Invoice table. It is up to the database programmer to ensure that linked records have matching primary and foreign keys.

Productivity Hint 26.2

Avoid Unnecessary Data Replication

It is very common for beginning database designers to replicate data in a table. Whenever you find yourself replicating data in a table, ask yourself if you can move the replicated data into a separate table and use a key such as a code or ID number to link the tables.

Consider this example from an Invoice table:

Invoice

...	Product_Code	Description	Price	...
...	CHAR (10)	CHAR (40)	DECIMAL (10, 2)	...
...	116-064	Toaster	24.95	...
...	116-064	Toaster	24.95	...
...	...	...	...	...

As you can see, some product information is replicated. Is this replication an error? It depends. The product description for the product with code 116-064 is always going to be "Toaster". Therefore, that correspondence should be stored in an external Product table.

The product price, however, can change over time. When it does, old invoices don't automatically use the new price. Thus, it makes sense to store the price that the customer was actually charged in an Invoice table. The current list price, however, is best stored in an external Product table.

26.1.3 Implementing One-to-Many Relationships

Each invoice is linked to exactly one customer. On the other hand, each invoice has many items. (As in Chapter 22, an item identifies the product, quantity, and unit price.) Thus, there is a *one-to-many* relationship between invoices and items.

In the C++ class, the `Item` objects are stored in a vector:

```
class Invoice
{
   ...
```

```
private:
   Customer* cust;
   vector<Item> items;
   double payment;
};
```

However, in a relational database, you need to store the information in tables. Surprisingly many programmers, when faced with this situation, commit a major blunder and replicate columns, one for each item, as in Figure 5.

Clearly, this design is not satisfactory. What should we do if there are more than three items on an invoice? Perhaps we should have ten columns instead? But that is

Invoice

Invoice_ Number	Customer_ Number	Product_ Code1	Quantity1	Product_ Code2	Quantity2	Product_ Code3	Quantity3	Payment
INTEGER	INTEGER	CHAR (10)	INTEGER	CHAR (10)	INTEGER	CHAR (10)	INTEGER	DECIMAL (10, 2)
11731	3175	116-064	3	257-535	1	643-119	2	0

Figure 5 A Poor Design for an Invoice Table with Replicated Columns

Invoice

Invoice_Number	Customer_Number	Payment
INTEGER	INTEGER	DECIMAL (10, 2)
11731	3175	0
11732	3176	249.50
11733	3175	0

Item

Invoice_Number	Product_Code	Quantity
INTEGER	CHAR (10)	INTEGER
11731	116-064	3
11731	257-535	1
11731	643-119	2
11732	116-064	10
11733	116-064	2
11733	643-119	1

Figure 6 Linked Invoice and Item Tables Implement a Multi-Valued Relationship

wasteful if the majority of invoices only have a couple of items, and it does not solve the problem of the occasional invoice with lots of items.

Instead, you should distribute the information into two tables: one for invoices and another for items. Link each item back to its invoice with an `Invoice_Number` foreign key in the item table—see Figure 6.

Our database now consists of four tables:

- Invoice
- Customer
- Item
- Product

Figure 7 shows the links between these tables. In the next section you will see how to query this database for information about invoices, customers, and products. The queries will take advantage of the links between the tables.

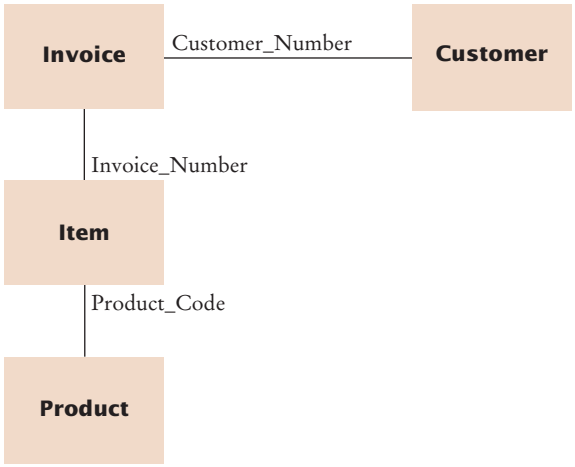

Figure 7 Links Between the Tables in the Sample Database

PRODUCTIVITY HINT 26.3

Don't Replicate Columns in a Table

If you find yourself numbering columns in a table with suffixes 1, 2, and so forth (such as `Quantity1`, `Quantity2`, `Quantity3` in the preceding example), then you are probably on the wrong track. How do you know there are exactly three quantities? In this case, it's time for another table.

Add a table to hold the information for which you replicated the columns. In that table, add a column that links back to a key in the first table, such as the invoice number in our example. By using an additional table, you can implement a one-to-many relationship.

26.2 Queries

Let us assume that the tables in our database have been created and that records have been inserted. Once a database is filled with data, you will want to query the database for information, such as

- What are the names and addresses of all customers?
- What are the names and addresses of all customers in California?
- What are the names and addresses of all customers who buy toasters?
- What are the names and addresses of all customers with unpaid invoices?

In this section you will learn how to formulate both simple and complex queries in SQL. We will use the data shown in Figure 8 for our examples.

Invoice

Invoice_Number	Customer_Number	Payment
INTEGER	INTEGER	DECIMAL (10, 2)
11731	3175	0
11732	3176	249.50
11733	3175	0

Item

Invoice_Number	Product_Code	Quantity
INTEGER	CHAR (10)	INTEGER
11731	116-064	3
11731	257-535	1
11731	643-119	2
11732	116-064	10
11733	116-064	2
11733	643-119	1

Product

Product_Code	Description	Unit_Price
CHAR (10)	CHAR (40)	DECIMAL (10, 2)
116-064	Toaster	24.95
257-535	Hair dryer	29.95
643-119	Car vacuum	19.99

Customer

Customer_Number	Name	Address	City	State	Zip
INTEGER	CHAR (40)	CHAR (40)	CHAR (30)	CHAR (2)	CHAR (10)
3175	Sam's Small Appliances	100 Main Street	Anytown	CA	98765
3176	Electronics Unlimited	1175 Liberty Ave	Pleasantville	MI	45066

Figure 8 A Sample Database

26.2.1 Simple Queries

Use the SQL SELECT command to query a database.

In SQL, you use the SELECT command to issue queries. For example, the command to select all data from the Customer table is

```
SELECT * FROM Customer
```

The outcome of the query is a view—a set of rows and columns that provides a "window" through which you can see some of the database data.

Customer_Number	Name	Address	City	State	Zip
3175	Sam's Small Appliances	100 Main Street	Anytown	CA	98765
3176	Electronics Unlimited	1175 Liberty Ave	Pleasantville	MI	45066

Many database systems have tools that allow you to issue interactive SQL commands—Figure 9 shows a typical example. When you issue a SELECT command, the tool displays the resulting view. You may want to skip ahead to Section 26.3 and install a database now—or see whether your computer lab has a database installed already. Then you can run the interactive SQL tool for your database and try out some queries.

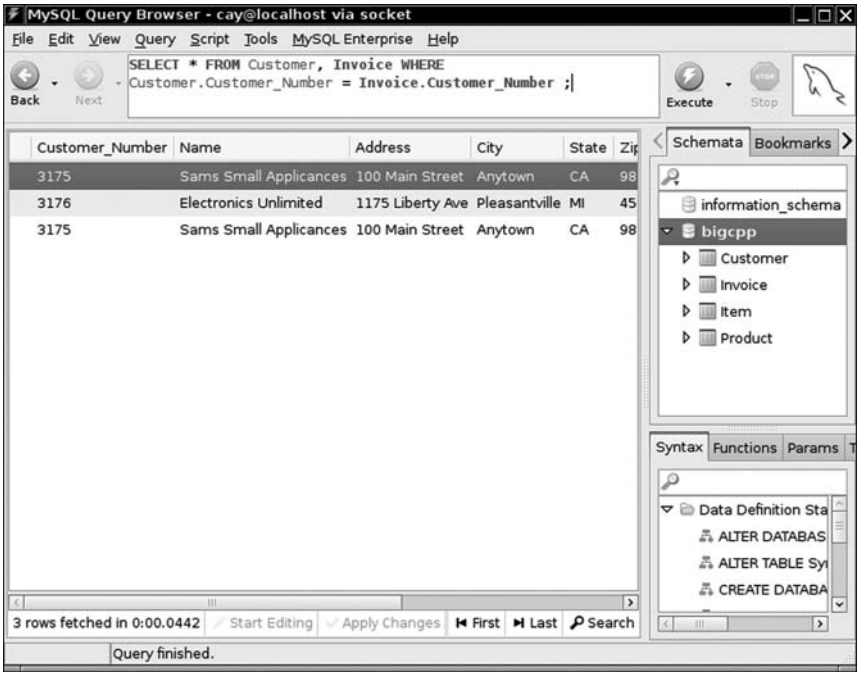

Figure 9 An Interactive SQL Tool

26.2.2 Selecting Columns

Often, you don't care about all columns in a table. Suppose your traveling salesperson is planning a trip to all customers. To plan the route, the salesperson just wants to know the cities and states of all customers. Here is the query:

```
SELECT City, State FROM Customer
```

The result is

City	State
Anytown	CA
Pleasantville	MI

As you can see, the syntax for selecting columns is straightforward. You simply specify the names of the columns you want, separated by commas.

26.2.3 Selecting Subsets

You just saw how you can restrict a view to show selected columns. Sometimes you want to select certain rows that fit a particular criterion. For example, you may want to find all customers in California. Whenever you want to select a subset, you use the `WHERE` clause, followed by the condition that describes the subset. Here is an example.

```
SELECT * FROM Customer WHERE State = 'CA'
```

The result is

Customer_Number	Name	Address	City	State	Zip
3175	Sam's Small Appliances	100 Main Street	Anytown	CA	98765

You have to be a bit careful with expressing the condition in the `WHERE` clause, because SQL syntax differs from the C++ syntax. As you already know, in SQL you use single quotes to delimit strings, such as `'CA'`. You also use a single `=`, not a double `==`, to test for equality. To test for inequality, you use the `<>` operator. For example

```
SELECT * FROM Customer WHERE State <> 'CA'
```

selects all customers that are not in California. (Whether or not string comparison is case sensitive depends, unfortunately, on the database vendor and the chosen installation defaults.)

You can match patterns with the `LIKE` operator. The right-hand side must be a string that can contain the special symbols `_` (match exactly one character) and `%` (match any character sequence). For example, the expression

```
Name LIKE '_o%'
```

matches all strings whose second character is an "o". Thus, "Toaster" is a match but "Crowbar" is not.

You can combine expressions with the logical connectives AND, OR, and NOT. (Do not use the C++ &&, ||, and ! operators.) For example,

```
SELECT *
   FROM Product
   WHERE Unit_Price < 100
      AND Description <> 'Toaster'
```

selects all products with a price less than 100 that are not toasters.

Of course, you can select both row and column subsets, such as

```
SELECT Name, City FROM Customer WHERE State = 'CA'
```

26.2.4 Calculations

Suppose you want to find out *how many* customers there are in California. Use the COUNT function:

```
SELECT COUNT(*) FROM Customer WHERE State = 'CA'
```

In addition to the COUNT function, there are four other functions: SUM, AVG (average), MAX, and MIN.

The * parameter of the COUNT function means that you want to calculate entire records. That is appropriate only for COUNT. For other functions, you have to access a specific column. Put the column name inside the parentheses:

```
SELECT AVG(Unit_Price) FROM Product
```

26.2.5 Joins

The queries that you have seen so far involve a single table. However, usually the information that you want is distributed over multiple tables. For example, suppose you are asked to find all invoices that include an item for a car vacuum. From the Product table, you can issue a query to find the product code:

```
SELECT Product_Code
   FROM Product
   WHERE Description = 'Car vacuum'
```

You will find out that the car vacuum has product code 643-119. Then you can issue a second query

```
SELECT Invoice_Number
   FROM Item
   WHERE Product_Code = '643-119'
```

But it makes sense to combine these two queries so that you don't have to keep track of the intermediate result. When combining queries, note that the two tables are linked by the Product_Code field. We want to look at matching rows in both tables. In other words, we want to restrict the search to rows where

```
Product.Product_Code = Item.Product_Code
```

Here, the syntax

TableName.*ColumnName*

denotes the column in a particular table. Whenever a query involves multiple tables, you should specify both the table name and the column name. Thus, the combined query is

```
SELECT Item.Invoice_Number
   FROM Product, Item
   WHERE Product.Description = 'Car vacuum'
      AND Product.Product_Code = Item.Product_Code
```

The result is

Invoice_Number
11731
11733

A *join* is a query that involves multiple tables.

In this query, the FROM clause contains the names of multiple tables, separated by commas. (It doesn't matter in which order you list the tables.) Such a query is often called a *join* because it involves joining multiple tables.

You may want to know in what cities hair dryers are popular. Now you need to add the Customer table to the query—it has the customer addresses. The customers are referenced by invoices, so you need that table as well. Here is the complete query:

```
SELECT Customer.City, Customer.State, Customer.Zip
   FROM Product, Item, Invoice, Customer
   WHERE Product.Description = 'Hair dryer'
      AND Product.Product_Code = Item.Product_Code
      AND Item.Invoice_Number = Invoice.Invoice_Number
      AND Invoice.Customer_Number
         = Customer.Customer_Number
```

The result is

City	State	Zip
Anytown	CA	98765

Whenever you formulate a query that involves multiple tables, remember to do all of the following:

- List all tables that are involved in the query in the FROM clause.
- Use the *TableName*.*ColumnName* syntax to refer to column names.
- List all join conditions (*TableName1*.*ColumnName1* = *TableName2*.*ColumnName2*) in the WHERE clause.

As you can see, these queries can get a bit complex. However, databases are very good at answering these queries (see Productivity Hint 26.5 on page 24). One

remarkable aspect of SQL is that you tell the database *what* you want, not *how* it should find the answer. It is entirely up to the database to come up with a plan for how to find the answer to your query in the shortest number of steps. Commercial database manufacturers take great pride in coming up with clever ways to speed up queries: query optimization strategies, caching of prior results, and so on. In this regard, SQL is a very different language from C++. SQL statements are descriptive and leave it to the database how to execute them. C++ statements are prescriptive—you spell out exactly the steps you want your program to carry out.

Common Error 26.1

Joining Tables Without Specifying a Link Condition

If you select data from multiple tables without a restriction, the result is somewhat surprising—you get a result set containing *all combinations* of the values, whether or not one of the combinations exists with actual data. For example, this query returns the following result set.

```
SELECT Invoice.Invoice_Number, Customer.Name
   FROM Invoice, Customer
```

Invoice.Invoice_Number	Customer.Name
11731	Sam's Small Appliances
11732	Sam's Small Appliances
11733	Sam's Small Appliances
11731	Electronics Unlimited
11732	Electronics Unlimited
11733	Electronics Unlimited

As you can see, the result set contains all six combinations of invoice numbers and customer names, even though three of those combinations don't occur with real invoices. You need to supply a `WHERE` clause to restrict the set of combinations; this query gets the result set below:

```
SELECT Invoice.Invoice_Number, Customer.Name
   FROM Invoice, Customer
   WHERE Invoice.Customer_Number = Customer.Customer_Number
```

Invoice.Invoice_Number	Customer.Name
11731	Sam's Small Appliances
11732	Electronics Unlimited
11733	Sam's Small Appliances

26.2.6 Updating and Deleting Data

The UPDATE and DELETE commands modify the data in a database.

Up to now, you have seen how to formulate increasingly complex SELECT queries. The outcome of a SELECT query is a *result set* that you can view and analyze. Two related statement types, UPDATE and DELETE, don't produce a result set. Instead, they modify the database.

The DELETE statement is the easier of the two. It simply deletes the rows that you specify. For example, to delete all customers in California, you issue the statement

```
DELETE FROM Customer WHERE State = 'CA'
```

The UPDATE query allows you to update columns of all records that fulfill a certain condition. For example, here is how you can add another unit to the quantity of every item in invoice number 11731.

```
UPDATE Item
   SET Quantity = Quantity + 1
   WHERE Invoice_Number = '11731'
```

You can update multiple column values by specifying multiple update expressions in the SET clause, separated by commas.

Both the DELETE and the UPDATE statements return a value, namely the number of rows that are deleted or updated.

26.3 Installing a Database

A wide variety of database systems are available. Among them are

- High-performance commercial databases such as Oracle, DB2, and Microsoft SQL Server
- Open-source databases such as PostgreSQL and MySQL
- Desktop databases such as Access and Foxpro

Which one should you choose for learning database programming? That depends greatly on your budget, computing resources, and experience with installing complex software. In a laboratory environment with a trained administrator, it makes a lot of sense to install a commercial database such as Oracle, but the products themselves, the hardware to run them, and the staff to administer them are expensive. Open-source alternatives are available free of charge, and their quality has greatly increased in recent years. Although they may not be suited for large-scale applications, they work fine for learning purposes. Desktop databases such as Access are less standard-compliant and more suitable for interactive use than for programming.

In this chapter, we will use the popular MySQL database. It is an open-source product that runs on a wide variety of platforms and is in use by many companies and organizations around the world. Of course, since SQL is a widely supported standard, you can use another database to try out the SQL commands of the preceding sections.

You use an application programming interface (API) to access a database from a C++ program.

However, connecting to a database from a C++ program is not standardized. Each database has its own API (application programming interface). Fortunately, after you learn the MySQL API, you should find it straightforward to learn another database API. The underlying principles are the same, even though the names of the individual functions and data structures differ.

If you work in a computing laboratory, someone will have installed a database for you, and you should be able to find instructions on how to use it. In this section, we discuss several concepts that you need to know when installing and administering your own database. The details depend on your operating system—see the companion Web site for more information.

Here we give you a general sequence of steps for installing a database and testing your installation.

1. Obtain the database program, either on CD-ROM or by downloading it from the Web.
2. Read the installation instructions.
3. Install the program. Details vary greatly. This may be as simple as running an installation program or as involved as recompiling the program from its source code.
4. Start the database. You need to start the database server before you can carry out any database operations. For example, with MySQL, the `mysqld` program starts the database server. Read the installation instructions for details.
5. Find out the name and password of the database administrator account. Sometimes, the database administrator account is the same as an administrator account for your operating system, but on other platforms, it is not. Read the installation instructions for details.
6. Locate the program for executing interactive SQL instructions. (With MySQL, the program is called `mysql`.) Find out how to log on as the database administrator. Then run the following SQL instructions:

   ```
   CREATE TABLE Test (Name CHAR(20))
   INSERT INTO Test VALUES ('Romeo')
   SELECT * FROM Test
   DROP TABLE Test
   ```

   At this point, you should get a display that shows a single row and column of the Test database, containing the string "Romeo". If not, carefully read the documentation of your SQL tool to see how you need to enter SQL statements. For example, with MySQL, you need a semicolon after each SQL statement.
7. Set up databases. A database is a collection of tables. A database program such as MySQL can manage multiple databases, each of which contains separate table sets and access permissions. This step typically involves running an administration program, logging in as database administrator, and creating one or more databases. For the programs in the next section, you need to create a database named "bigcpp".

8. Set up user accounts. This step typically involves running an administration program, logging in as database administrator, and adding user names, passwords, and permissions. If you are the only user of the database, you may be able to use a default account. Again, details vary greatly among databases, and you should consult the documentation.

Once your database is set up and functioning correctly, you need to find out how to connect to it from C++. The preceding instructions applied to all SQL databases, but the following steps are specific to MySQL.

1. Make a copy of the `execsql.cpp` program from Section 27.4.7. Examine the call to the `mysql_real_connect` function inside the `main` function:

   ```
   if (mysql_real_connect(connection, NULL, NULL, NULL,
      "bigcpp", NULL, 0, NULL) == NULL)
   ```

   If you need to connect to a remote database or a database with multiple users, change the first three `NULL` parameters to strings denoting the host server, the database user name, and the password.

2. Locate the directories for the header file `mysql.h` and the `mysqlclient` library files. They should be in the `include` and `lib` subdirectories of the MySQL installation directory. Add the path of the header file directory to your compiler's header file directories. Add the library to your compiler's libraries. Compile the `execsql` program. For example, on Linux, you use the command

   ```
   g++ -o execsql `mysql_config --cflags` execsql.cpp
      `mysql_config --libs`
   ```

3. Locate the file `test.sql` that contains the same SQL instructions as the database test in Step 6 above. It is included with the code distribution for this book. Then run the command

   ```
   execsql < test.sql
   ```

   The program should print "Romeo", the same output as with the first database test. If this test passes, then you are ready to run the programs of the next section.

   If your program doesn't work, there are several possible causes.

   - If the `execsql` program didn't compile, check the settings for the header files and the libraries. Make sure that `mysql.h` and the `mysqlclient` library are included.
   - If the program starts and does nothing, check that you invoked it correctly, as `execsql < test.sql`.
   - If the program exits with an error message, check that you started the database before running the `execsql` program.
   - Check that you created the `bigcpp` database and gave at least one database user the right to access it.
   - If you left the `mysql_real_connect` parameters as `NULL`, check that the user running the program has the right to access the database.

Productivity Hint 26.4

Looking for Help on the Internet

When setting up a new software package, particularly one as complex as a database, it is extremely common to make a few errors. Fortunately, advice for many problems can be readily found on the Internet. Software providers, user groups, and helpful individuals have set up Web sites with installation directions, FAQ (frequently asked question) lists, discussion boards, and tutorials.

If you encounter an incomprehensible error message, try pasting it into your favorite search engine. For example, suppose you try to compile the `execsql` program under Linux and get the error message: "my_compress.o: undefined reference to compress". An Internet search will reveal many discussion groups with pleas from help by other programmers who encountered the exact same problem. You might want to join one of the discussion groups. But before you do and post your own plea for help, first read some of the answers. In this case, you will find a suggestion to add the `-lz` option, which links in the compression library. Try it, and the error message goes away.

The Internet is a marvelous tool for support. Instead of toiling in isolation, you can learn from the experience of your peers. However, be sure to do your research first rather than making others do your work—see the excellent article [2] by Eric Raymond and Rick Moen.

Random Fact 26.1

Open Source and Free Software

Most companies that produce software regard the source code as a trade secret. After all, if customers or competitors had access to the source code, they could study it and create similar programs without paying the original vendor. For the same reason, customers dislike secret source code. If a company goes out of business or decides to discontinue support for a computer program, its users are left stranded. They are unable to fix bugs or adapt the program to a new operating system.

Nowadays, some software packages are distributed with "open source" or "free software" licenses. Here, the term "free" doesn't refer to price, but to the freedom to inspect and modify the source code. Richard Stallman, a famous computer scientist and winner of a MacArthur "genius" grant, pioneered the concept of free software. He is the inventor of the Emacs text editor and the originator of the GNU project that aims to create an entirely free version of a Unix compatible operating system. All programs of the GNU project are licensed under the General Public License or GPL. The GPL allows you to make as many copies as you wish, make any modifications to the source, and redistribute the original and modified programs, charging nothing at all or whatever the market will bear. In return, you must agree that your modifications also fall under the GPL. You must give out the source code to any changes that you distribute, and anyone else can distribute them under the same conditions. The GPL, and similar open source licenses, form a social contract. Users of the software enjoy the freedom to use and modify the software, and in return they are obligated to share any improvements that they make. Many programs, such as the Linux operating system and the MySQL database, are distributed under the GPL.

Some commercial software vendors have attacked the GPL as "viral" and "undermining the commercial software sector". Other companies have a more nuanced strategy, producing

proprietary software while also contributing to open source projects. MySQL, in particular, can be licensed from its vendor under a non-GPL license so that it can be included in proprietary software. Of course, then the licensee needs to share the profits with the vendor.

Frankly, open source is not a panacea and there is plenty of room for the commercial software sector. Open source software often lacks the polish of commercial software because many of the programmers are volunteers who are interested in solving their own problems, not in making a product that is easy to use by others. Some product categories are not available at all as open source software because the development work is unattractive when there is little promise of commercial gain. Open source software has been most successful in areas that are of interest to programmers, such as the Linux operating system, Web servers, and programming tools.

On the positive side, the open software community can be very competitive and creative. It is quite common to see several competing projects that take ideas from each other, all rapidly becoming more capable. Having many programmers involved, all reading the source code, means that bugs tend to get squashed quickly. Eric Raymond describes open source development in his famous article "The Cathedral and the Bazaar" [3]. He writes "Given enough eyeballs, all bugs are shallow".

26.4 Database Programming in C++

26.4.1 Connecting to the MySQL Database

The MySQL database includes an application programming interface (API) that allows C and C++ programs to access the database, execute database commands, issue queries, and retrieve the query results. Because the API is usable with the C programming language, it does not use classes, and it uses character arrays instead of strings. We will simply convert between C++ `string` objects and character arrays as necessary. (There is also a separate C++ API for MySQL, but it is more complex, and we do not discuss it here.)

A database program runs continuously, waiting for requests from other programs. When you write your own program that interacts with the database, your program needs to establish a *connection* to the database. The connection is the route for the database commands that your program sends and the answers that the database sends back.

To connect to a MySQL database, first call `mysql_init` to obtain an object that encapsulates the connection. You get a pointer that you need to save.

```
MYSQL* connection = mysql_init();
```

Next, call the `mysql_real_connect` function. You supply eight arguments:

- A pointer to an initialized connection object
- Three strings with the host name, database user name, and password, or `NULL` to use the defaults

- The name of the database that you want to use. We recommend that you do all your work in a database named `bigcpp`.
- A set of three specialized parameters that you should leave at their defaults (`0`, `NULL`, `0`) unless a system administrator tells you otherwise

Typically, the call looks like this:

```
mysql_real_connect(connection, NULL, NULL, NULL,
   "bigcpp", 0, NULL, 0)
```

You need to check the return value of this call. If the function returns `NULL`, there was a connection error. At this stage, errors are quite common. For example, if the database is not running, the connection will fail. When there is an error with a database function, call the `mysql_error` function to retrieve a description of the last error. For example:

```
if (mysql_real_connect(...) == NULL)
{
   string error_message = mysql_error(connection);
   ...
}
```

At the end of the database program, you need to close the connection:

```
mysql_close(connection);
```

This is an important step. Not only do you recycle the memory for the connection object, you also notify the database that it can free the data structures that it set up to communicate with your program. Be sure to call the `mysql_close` function at the end of your program, even if your program terminates because of an error.

26.4.2 Executing SQL Statements

Once you are connected to the database, you can issue commands. Assemble the command in a `string` object. Use the `c_str` member function to convert the `string` object to a C string. For example,

```
string new_value = ...;
string query = "INSERT INTO Test VALUES ('" + new_value + "')";
mysql_query(query.c_str());
```

If a command depends on a number, you need to convert the number to a string, for example with the `int_to_string` and `double_to_string` functions of Chapter 9.

```
int qty = ...;
string command = "SELECT * FROM Item WHERE Quantity >= "
   + int_to_string(qty);
mysql_query(connection, command.c_str());
```

Note that you use the `mysql_query` function both for queries and updates. You will see in the next section how you obtain the result of a query.

If the command is not a legal SQL command, the `mysql_query` function returns an error code. If the command was executed correctly, the return value is `0`. Because

it is common to accidentally create syntax errors in command strings, you should always check the return value:

```
if (mysql_query(connection, command.c_str()) != 0)
{
   cout << "Error: " << mysql_error(connection) << "\n";
   ...
}
```

Do not use the query result if the `mysql_query` function returns an error code.

26.4.3 Analyzing Query Results

After issuing a query, you load the result into a `MYSQL_RES` object:

```
MYSQL_RES* result = mysql_store_result(connection);
```

If the preceding command was a `SELECT` query, then a result object is allocated and filled with the selected data. If the preceding command was not a query, the `mysql_store_result` function returns a `NULL` pointer.

The `mysql_num_rows` and `mysql_num_fields` functions return the number of rows and columns of the data set:

```
int rows = mysql_num_rows(result);
int fields = mysql_num_fields(result);
```

If `rows` is zero, then the query yielded no answer. To analyze the query result, you need to fetch each row. Calling `mysql.fetch_row` fetches the next row from the result set. Call that function `rows` times:

```
for (int r = 1; r <= rows; r++)
{
   MYSQL_ROW row = mysql_fetch_row(result);
   inspect field data from the current row
}
```

A `MYSQL_ROW` is an array of C-style strings, containing the fields of the row. It is best to convert them immediately to C++ `string` objects. For example, if `i` is a number between `0` and `fields - 1`, then the `i`th field can be obtained as

```
string field = row[i];
```

If you need the field as a number, use functions such as the `string_to_int` or `string_to_double` of Chapter 9 to convert the strings to numbers. For example, suppose you issued a query of the form

```
SELECT Unit_Price, ... FROM Product WHERE ...
```

Then you get the price as

```
double price = string_to_double(row[0]);
```

When you are done using a `MYSQL_RESULT`, you should close it:

```
mysql_free_result(result);
```

As you can see, it is quite easy to analyze the result of a database query. Keep in mind that this mechanism is designed for analyzing query results of moderate size.

You should not query the database for huge result sets and then process them in C++—see Productivity Hint 26.5 on page 24.

The `execsql` program puts these concepts to use. It simply executes a number of commands that are stored in a text file. For example, suppose you have a text file `product.sql` with SQL commands to populate the Product table.

Then run the program as

```
execsql < product.sql
```

The program executes the statements in the text file and prints out the result of the `SELECT` query.

You can also use the program as an interactive testing tool. Run

```
execsql
```

without any command-line parameters. Type in SQL commands at the command line. Every time you hit the Enter key, the command is executed.

ch26/execsql.cpp

```
#include <iostream>

#include <string>
#include <mysql.h>

using namespace std;

void execute_command(MYSQL* connection, string command)
{
   if (mysql_query(connection, command.c_str()) != 0)
   {
      cout << "Error: " << mysql_error(connection) << "\n";
      return;
   }

   MYSQL_RES* result = mysql_store_result(connection);
   if (result == NULL) return;

   int rows = mysql_num_rows(result);
   int fields = mysql_num_fields(result);
   for (int r = 1; r <= rows; r++)
   {
      MYSQL_ROW row = mysql_fetch_row(result);
      for (int f = 0; f < fields; f++)
      {
         string field(row[f]);
         if (f > 0) cout << ",";
         cout << field;
      }
      cout << "\n";
   }

   mysql_free_result(result);
}

```

```
36 int main()
37 {
38    MYSQL* connection = mysql_init(NULL);
39
40    if (mysql_real_connect(connection, NULL, NULL, NULL,
41          "bigcpp", 0, NULL, 0) == NULL)
42    {
43       cout << "Error: " << mysql_error(connection) << "\n";
44       return 1;
45    }
46
47    string line;
48    bool more = true;
49    while (getline(cin, line))
50    {
51       execute_command(connection, line);
52    }
53
54    mysql_close(connection);
55    return 0;
56 }
```

ch26/product.sql

```
1 CREATE TABLE Product (Product_Code CHAR(10), Description CHAR(40),
2    Unit_Price DECIMAL(10, 2))
3 INSERT INTO Product VALUES ('116-064', 'Toaster', 24.95)
4 INSERT INTO Product VALUES ('257-535', 'Hair dryer', 29.95)
5 INSERT INTO Product VALUES ('643-119', 'Car vacuum', 19.99)
6 SELECT * FROM Product
```

PRODUCTIVITY HINT 26.5

Let the Database Do the Work

You now know how to issue a SQL query from a C++ program and iterate through the result set. A common error that students make is to iterate through one table at a time to find a result. For example, suppose you want to find all invoices that contain car vacuums. You could use the following plan:

1. Issue the query `SELECT * FROM Product` and iterate through the result set to find the product code for a car vacuum.
2. Issue the query `SELECT * FROM Item` and iterate through the result set to find the items with that product code.

However, this plan is *extremely inefficient*. Such a program does in very slow motion what a database has been designed to do quickly. Instead, let the database do all the work. Give the complete query to the database:

```
SELECT Item.Invoice_Number
   FROM Product, Item
   WHERE Product.Description = 'Car vacuum'
      AND Product.Product_Code = Item.Product_Code
```

Then iterate through the result set to read off all invoice numbers.

Beginners are often afraid of issuing complex SQL queries. However, you are throwing away a major benefit of a relational database if you don't take advantage of SQL.

26.5 Case Study: Accessing an Invoice Database

In this section, we will develop two database programs in C++. The first program queries the invoice database developed in Section 26.1. The user supplies an invoice number, and the program prints the invoice. The second program can be used to add a new invoice to the database.

Because we need conversion functions such as `string_to_double` for both programs, we placed them in a separate file `sutil.cpp` and included the header file `sutil.h`. These files are included with the companion code for this book.

To print an invoice, we first construct a query to obtain the customer number and payment for a given invoice number (stored in the string variable `invnum`).

```
string query = "SELECT Customer_Number, Payment FROM Invoice "
   "WHERE Invoice_Number = '" + invnum + "'";
```

Note how a long query string can be broken up over multiple lines. In C++, adjacent quoted strings are automatically concatenated. (You cannot use the `+` operator to combine two literal strings because `+` is not defined if both arguments are of type `char*`.)

We issue the query. If there are no rows in the result, then there is no matching invoice. Otherwise, we fetch the first row and extract the customer number and payment.

```
MYSQL_ROW row = mysql_fetch_row(result);
string custnum = row[0];
double payment = string_to_double(row[1]);
```

Note that the query specifically asks for the customer number and payment. We prefer specific queries to queries of the form `SELECT *`. If a database is reorganized and columns are added or rearranged, then the C++ code that analyzes the result of a `SELECT *` query may look for fields in the wrong positions.

Once we have the customer number, it is a simple matter to query and print the customer data. You will find the details in the `print_customer` function. Printing the items is a bit more complex because we want to print the product descriptions and prices, not the product codes. The query links the Item and Product tables:

```
string query = "SELECT Item.Quantity, Product.Description, "
   "Product.Unit_Price FROM Item, Product WHERE "
   "Item.Invoice_Number = ' "+ invnum
   + "'AND Item.Product_Code = Product.Product_Code";
```

Finally, we want to print the amount due. We let the database compute the sum of quantities and unit prices:

```
string query = "SELECT SUM(Item.Quantity * Product.Unit_Price) "
   "FROM Item, Product WHERE Item.Invoice_Number = '"
   + invnum + "' AND Item.Product_Code = Product.Product_Code";
```

The result is a table with a single row and column. Fetch the field value and convert it to a floating-point number:

```
MYSQL_ROW row = mysql_fetch_row(result);
double amount_due = string_to_double(row[0]);
```

You will find the complete program at the end of this section. As you can see, most of the program consists of assembling queries, moving data out of result sets, and checking for database errors.

The second sample program allows users to add new invoices to the database. The program prompts for the customer number and payment. (Exercise P26.9 suggests a useful enhancement: to search for the customer number if it is not known.) Then the program needs to assign an invoice number that is different from all other invoice numbers. We first find the maximum of all invoice numbers by issuing the query

```
SELECT MAX(Invoice_Number) FROM Invoice
```

Then we add 1 to the maximum in order to obtain a new invoice number. We insert a new row into the database:

```
string command = "INSERT INTO Item VALUES ('" + invnum +
   "', '" + prodcode + "', " + int_to_string(quantity) + ")";
```

The remainder of the program is straightforward. The program simply prompts for product codes and quantities and adds rows to the Items table.

We kept these programs simple so that you can focus on the code that is required to interact with the database. Exercise P26.10 suggests a more object-oriented approach in which you convert between C++ classes, such as `Invoice` and `Customer`, and the relational data in the database.

ch26/printinv.cpp

```
1  #include <iostream>
2  #include <iomanip>
3
4  #include <string>
5  #include <mysql.h>
6
7  #include "sutil.h"
8
9  using namespace std;
10
11 /**
12    Prints a customer with a given customer number.
13    @param connection the database connection
14    @param custnum the customer number
15 */
16 void print_customer(MYSQL* connection, string custnum)
17 {
```

```
   string query = "SELECT Name, Address, City, State, Zip "
      "FROM Customer WHERE Customer_Number = '" + custnum + "'";
   if (mysql_query(connection, query.c_str()) != 0)
   {
      cout << "Error: " << mysql_error(connection) << "\n";
      return;
   }
   MYSQL_RES* result = mysql_store_result(connection);
   if (result == NULL)
   {
      cout << "Error: " << mysql_error(connection) << "\n";
      return;
   }
   int rows = mysql_num_rows(result);
   if (rows == 0)
   {
      cout << "Customer not found.\n";
      return;
   }

   MYSQL_ROW row = mysql_fetch_row(result);
   string name = row[0];
   string street = row[1];
   string city = row[2];
   string state = row[3];
   string zip = row[4];
   mysql_free_result(result);

   cout << name << "\n" << street << "\n"
      << city << ", " << state << " " << zip << "\n";
}

/**
   Prints all items of an invoice.
   @param connection the database connection
   @param invnum the invoice number
*/
void print_items(MYSQL* connection, string invnum)
{
   string query = "SELECT Item.Quantity, Product.Description, "
     "Product.Unit_Price FROM Item, Product WHERE Item.Invoice_Number ='"
      + invnum + "'AND Item.Product_Code = Product.Product_Code";
   if (mysql_query(connection, query.c_str()) != 0)
   {
      cout << "Error: " << mysql_error(connection) << "\n";
      return;
   }
   MYSQL_RES* result = mysql_store_result(connection);
   if (result == NULL)
   {
      cout << "Error: " << mysql_error(connection) << "\n";
      return;
  }
   int rows = mysql_num_rows(result);
```

```
72
73       const int COLUMN_WIDTH = 30;
74
75       cout <<
76          "\n\nDescription                    Price  Qty  Total\n";
77
78       for (int r = 1; r <= rows; r++)
79       {
80          MYSQL_ROW row = mysql_fetch_row(result);
81          int quantity = string_to_int(row[0]);
82          string description = row[1];
83          double price = string_to_double(row[2]);
84
85          cout << description;
86
87          // Pad with spaces to fill column
88
89          int pad = COLUMN_WIDTH - description.length();
90          for (int i = 1; i <= pad; i++)
91             cout << " ";
92
93          cout << price
94             << "   " << quantity
95             << "   " << price * quantity << "\n";
96       }
97
98       mysql_free_result(result);
99    }
100
101   /**
102      Gets the amount due on all items of an invoice.
103      @param connection the database connection
104      @param invnum the invoice number
105   */
106   double get_amount_due(MYSQL* connection, string invnum)
107   {
108      string query = "SELECT SUM(Item.Quantity * Product.Unit_Price) "
109         "FROM Item, Product WHERE Item.Invoice_Number = '"
110         + invnum + "' AND Item.Product_Code = Product.Product_Code";
111      if (mysql_query(connection, query.c_str()) != 0)
112      {
113         cout << "Error: " << mysql_error(connection) << "\n";
114         return 0;
115      }
116      MYSQL_RES* result = mysql_store_result(connection);
117      if (result == NULL)
118      {
119         cout << "Error: " << mysql_error(connection) << "\n";
120         return 0;
121      }
122      int rows = mysql_num_rows(result);
123      if (rows == 0) return 0;
124      MYSQL_ROW row = mysql_fetch_row(result);
125      double amount_due = string_to_double(row[0]);
```

```
126    mysql_free_result(result);
127    return amount_due;
128 }
129
130 /**
131    Prints an invoice.
132    @param connection the database connection
133    @param invnum the invoice number
134 */
135 void print_invoice(MYSQL* connection, string invnum)
136 {
137    string query = "SELECT Customer_Number, Payment FROM Invoice "
138       "WHERE Invoice_Number = '" + invnum + "'";
139    if (mysql_query(connection, query.c_str()) != 0)
140    {
141       cout << "Error: " << mysql_error(connection) << "\n";
142       return;
143    }
144    MYSQL_RES* result = mysql_store_result(connection);
145    if (result == NULL)
146    {
147       cout << "Error: " << mysql_error(connection) << "\n";
148       return;
149    }
150    int rows = mysql_num_rows(result);
151    if (rows == 0)
152    {
153       cout << "Invoice not found.\n";
154       return;
155    }
156
157    MYSQL_ROW row = mysql_fetch_row(result);
158    string custnum = row[0];
159    double payment = string_to_double(row[1]);
160    mysql_free_result(result);
161
162    cout << "                    I N V O I C E\n\n";
163
164    print_customer(connection, custnum);
165    print_items(connection, invnum);
166
167    double amount_due = get_amount_due(connection, invnum);
168
169    cout << "\nAMOUNT DUE: $" << amount_due - payment << "\n";
170 }
171
172 int main()
173 {
174    MYSQL* connection = mysql_init(NULL);
175
176    if (mysql_real_connect(connection, NULL, NULL, NULL,
177          "bigcpp", 0, NULL, 0) == NULL)
178    {
179       cout << "Error: " << mysql_error(connection) << "\n";
```

```
180       return 1;
181    }
182
183    cout << "Enter invoice number: ";
184    string invnum;
185
186    getline(cin, invnum);
187    print_invoice(connection, invnum);
188
189    mysql_close(connection);
190    return 0;
191 }
```

ch26/addinv.cpp

```
1  #include <iostream>
2  #include <iomanip>
3
4  #include <string>
5  #include <mysql.h>
6
7  #include "sutil.h"
8
9  using namespace std;
10
11 /**
12    Finds a customer with a given customer number.
13    @param connection the database connection
14    @param custnum the customer number
15    @return true if a customer with the given number exists
16 */
17 bool find_customer(MYSQL* connection, string custnum)
18 {
19    string query = "SELECT * FROM Customer WHERE Customer_Number = '"
20       + custnum + "'";
21    if (mysql_query(connection, query.c_str()) != 0)
22    {
23       cout << "Error: " << mysql_error(connection) << "\n";
24       return false;
25    }
26    MYSQL_RES* result = mysql_store_result(connection);
27    if (result == NULL)
28    {
29       cout << "Error: " << mysql_error(connection) << "\n";
30       return false;
31    }
32    bool r = mysql_num_rows(result) > 0;
33    mysql_free_result(result);
34    return r;
35 }
36
```

```
37  /**
38     Finds a product with a given product code.
39     @param connection the database connection
40     @param prodcode the product code
41     @return true if a product with the given code exists
42  */
43  bool find_product(MYSQL* connection, string prodcode)
44  {
45     string query = "SELECT * FROM Product WHERE Product_Code = '"
46        + prodcode + "'";
47     if (mysql_query(connection, query.c_str()) != 0)
48     {
49        cout << "Error: " << mysql_error(connection) << "\n";
50        return false;
51     }
52     MYSQL_RES* result = mysql_store_result(connection);
53     if (result == NULL)
54     {
55        cout << "Error: " << mysql_error(connection) << "\n";
56        return false;
57     }
58     bool r = mysql_num_rows(result) > 0;
59     mysql_free_result(result);
60     return r;
61  }
62
63  /**
64     Adds an invoice to the database.
65     @param connection the database connection
66     @param custnum the customer number
67     @param payment the payment amount
68     @return the automatically assigned invoice number
69  */
70  string add_invoice(MYSQL* connection, string custnum, double payment)
71  {
72     string query = "SELECT MAX(Invoice_Number) FROM Invoice";
73     if (mysql_query(connection, query.c_str()) != 0)
74     {
75        cout << "Error: " << mysql_error(connection) << "\n";
76        return "";
77     }
78     MYSQL_RES* result = mysql_store_result(connection);
79     if (result == NULL)
80     {
81        cout << "Error: " << mysql_error(connection) << "\n";
82        return "";
83     }
84     int rows = mysql_num_rows(result);
85     if (rows == 0) return "";
86     MYSQL_ROW row = mysql_fetch_row(result);
87     int max = string_to_int(row[0]);
88     mysql_free_result(result);
89
```

```
90       string invnum = int_to_string(max + 1);
91       string command = "INSERT INTO Invoice VALUES ('" + invnum + "', '"
92          + custnum + "', " + double_to_string(payment) + ")";
93       if (mysql_query(connection, command.c_str()) != 0)
94       {
95          cout << "Error: " << mysql_error(connection) << "\n";
96          return "";
97       }
98       return invnum;
99    }
100
101   /**
102      Adds an item to the database.
103      @param connection the database connection
104      @param invnum the invoice number
105      @param prodcode the product code
106      @param quantity the quantity
107   */
108   void add_item(MYSQL* connection, string invnum, string prodcode,
109      int quantity)
110   {
111      string command = "INSERT INTO Item VALUES ('" + invnum + "', '"
112         + prodcode + "', " + int_to_string(quantity) + ")";
113      if (mysql_query(connection, command.c_str()) != 0)
114      {
115         cout << "Error: " << mysql_error(connection) << "\n";
116         return;
117      }
118   }
119
120   int main()
121   {
122      MYSQL* connection = mysql_init(NULL);
123
124      if (mysql_real_connect(connection, NULL, NULL, NULL,
125            "bigcpp", 0, NULL, 0) == NULL)
126      {
127         cout << "Error: " << mysql_error(connection) << "\n";
128         return 1;
129      }
130
131      cout << "Enter customer number: ";
132      string custnum;
133      cin >> custnum;
134
135      if (!find_customer(connection, custnum))
136      {
137         cout << "Customer not found.\n";
138         mysql_close(connection);
139         return 0;
140      }
141
142      cout << "Enter payment: ";
143      double payment;
144      cin >> payment;
```

```
145
146     string invnum = add_invoice(connection, custnum, payment);
147     if (invnum == "")
148     {
149        mysql_close(connection);
150        return 0;
151     }
152
153     bool more = true;
154     while (more)
155     {
156        cout << "Enter product code, - when done: ";
157        string prodcode;
158        cin >> prodcode;
159        if (prodcode == "-") more = false;
160        else
161        {
162           if (find_product(connection, prodcode))
163           {
164              cout << "Enter quantity: ";
165              int quantity;
166              cin >> quantity;
167              add_item(connection, invnum, prodcode, quantity);
168           }
169           else cout << "Product not found.\n";
170        }
171     }
172     cout << "Added invoice " << invnum << "\n";
173     mysql_close(connection);
174     return 0;
175  }
```

ADVANCED TOPIC 26.1

Transactions

A transaction is a set of database updates that should either succeed in its entirety or not at all.

An important part of database processing is *transaction handling*. A transaction is a set of database updates that should either succeed in their entirety or not happen at all. For example, consider a banking application that transfers money from one account to another. This operation involves two steps: reducing the balance of one account and increasing the balance of another account. No software system is perfect, and there is always the possibility of an error. The banking application, the database program, or the network connection between them could exhibit an error right after the first part—then the money would be withdrawn from the first account but never deposited to the second account. Clearly, this would be very bad. There are many other similar situations. For example, if you change an airline reservation, you don't want to give up your old seat until the new one is confirmed.

What all these situations have in common is that there is a set of database operations that are grouped together to carry out the transaction. All operations in the group must be carried out together—a partial completion cannot be tolerated. In SQL, you use the `COMMIT` and

ROLLBACK commands to manage transactions. For example, to transfer money from one account to another, issue the commands

```
UPDATE Account SET Balance = Balance - 1000
   WHERE Account_Number = '95667-2574'
UPDATE Account SET Balance = Balance + 1000
   WHERE Account_Number = '82041-1196'
COMMIT
```

The COMMIT command makes the updates permanent. Conversely, the ROLLBACK command undoes all changes up to the last COMMIT.

You may wonder how a database can undo updates when a transaction is rolled back. The database actually stores your changes in a set of temporary tables. If you make queries within a transaction, the information in the temporary tables is merged with the permanent data for the purpose of computing the query result, giving you the illusion that the updates have already taken place. When you commit the transaction, the temporary data are made permanent. When you execute a rollback, the temporary tables are simply discarded.

Another database integrity issue arises from the fact that multiple users may access the data at the same time. Suppose for example that two users connect to the Invoice database of the preceding chapter at the same time. Each of them adds an invoice. Each of their programs queries the maximum of the invoice numbers, getting the same value. Each of them increments the maximum to get the next invoice number. Now both new invoices have the same number. You can solve this issue by making the computation of the new invoice number a part of the transaction, and instructing the database to isolate transactions that belong to different users.

One of the criteria for the reliability of a database is the ACID test. ACID is an acronym for the following four concepts:

- Atomicity: Either all steps of a transaction are executed or none of them are.
- Consistency: If a value is stored in multiple locations, it is either changed in all of them or none of them.
- Isolation: Concurrent transactions do not interfere with another.
- Durability: If the system fails and is restarted, all data reverts to the state of the last committed transaction.

If you take a class in database programming, you will encounter these concepts again in much greater detail. At this point, we hope that this note made you appreciate that real-world database programming is quite a bit more complex than issuing a few SQL queries.

CHAPTER SUMMARY

1. A relational database stores information in tables. Each table column has a name and a data type.
2. SQL (Structured Query Language) is a command language for interacting with a database.
3. Use the SQL commands CREATE TABLE and INSERT INTO to add data to a database.

4. You should avoid rows with replicated data. Instead, distribute the data over multiple tables.
5. A primary key is a column (or set of columns) whose value uniquely specifies a table record.
6. A foreign key is a reference to a primary key in a linked table.
7. Use the SQL `SELECT` command to query a database.
8. A *join* is a query that involves multiple tables.
9. The `UPDATE` and `DELETE` commands modify the data in a database.
10. You use an application programming interface (API) to access a database from a C++ program.
11. A transaction is a set of database updates that should either succeed in its entirety or not at all.

FURTHER READING

1. Chris J. Date and Hugh Darwen, *A Guide to the SQL Standard: A User's Guide to the Standard Database Language SQL*, Addison-Wesley, 1997.
2. Eric Raymond and Rick Moen, "How to Ask Questions the Smart Way", **`www.catb.org/~esr/faqs/smart-questions.html`**.
3. Eric Raymond, "The Cathedral and the Bazaar", **`www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/index.html`**.

REVIEW EXERCISES

Exercise R26.1. Design a set of database tables to store people and cars. A person has a name, a unique driver's license number, and an address. Every car has a unique vehicle identification number, manufacturer, type, and year. Every car has one owner, but one person can own multiple cars.

Exercise R26.2. Design a set of database tables to store library books and patrons. A book has an ISBN (International Standard Book Number), an author, and a title. The library may have multiple copies of each book, each with a different book ID. A patron has a name, a unique ID, and an address. A book may be checked out by at most one patron, but one patron can check out multiple books.

Exercise R26.3. Design a set of database tables to store sets of problems in a quiz. Each quiz has a title and a unique ID. Each problem has a unique ID, a question, and an answer. Each quiz contains a collection of problems.

Exercise R26.4. Design a set of database tables to store students, classes, professors, and classrooms. Each student takes zero or more classes. Each class has one professor, but a professor can teach multiple classes. Each class has one classroom.

Exercise R26.5. Give SQL commands to create a Book table, with columns for the ISBN, author, and title, and to insert all textbooks that you are using this semester.

Exercise R26.6. Give SQL commands to create a Car table, with columns for the vehicle identification number, manufacturer, type, and year of cars, and to insert all cars that your family members own.

Exercise R26.7. Give a SQL query that lists all products in the Invoice database.

Exercise R26.8. Give a SQL query that lists all customers in California.

Exercise R26.9. Give a SQL query that lists all customers in California or Nevada.

Exercise R26.10. Give a SQL query that lists all customers not in Hawaii.

Exercise R26.11. Give a SQL query that lists all customers who have an unpaid invoice.

Exercise R26.12. Give a SQL query that lists all products that have been purchased by a customer in California.

Exercise R26.13. Give a SQL query that lists all items that are part of invoice number 11731.

Exercise R26.14. Give a SQL query that computes the sum of all quantities that are part of invoice number 11731.

Exercise R26.15. Give a SQL query that computes the total cost. `SUM(Product.Unit_Price * Item.Quantity)` of all items in invoice number 11731.

Exercise R26.16. Give a SQL update statement that raises all prices by 10 percent.

Exercise R26.17. Give a SQL statement that deletes all customers in California.

PROGRAMMING EXERCISES

Exercise P26.1. Write a C++ program that creates a Car table with fields for the car manufacturer, type, model year, and fuel efficiency rating. Insert several cars. Print out the average fuel efficiency. Use `CREATE TABLE`, `INSERT`, and `SELECT AVG` SQL commands.

Exercise P26.2. Improve the `execsql` program and make the columns of the output line up. *Hint:* Read the description of the `mysql_fetch_lengths` function in the MySQL manual.

Exercise P26.3. Reimplement the employee data program of Section 9.6 using a SQL database.

Exercise P26.4. Write a C++ program that uses the database tables from the Invoice database. Produce a report that lists all customers, their invoices, the amounts paid, and the unpaid balances.

Exercise P26.5. Write a C++ program that uses a library database of books and patron data, as described in Exercise R26.2. Patrons should be able to check out and return books. Supply commands to print the books that a patron has checked out and to find out who has checked out a particular book. You may create and populate Patron and Book tables before running the program.

Exercise P26.6. Write a C++ program that creates a grade book for a class. You may create and populate a Student table and other tables that you need before running the program. The program should be able to display all grades for a given student. It should allow the instructor to add a new grade (such as "Homework 4: 100") or modify an existing grade.

Exercise P26.7. Write a program that assigns seats on an airplane as described in Exercise P22.7. Keep the passenger and seating information in a database.

Exercise P26.8. Write a program that keeps an appointment calendar in a database. Follow the description of Exercise P22.6.

Exercise P26.9. Enhance the program for adding an invoice so that a user can search the database for customers and products instead of just entering the customer number and product code. The user should be allowed to enter a part of the customer name or product description. Display all matching records and allow the user to select one of them.

Exercise P26.10. Reimplement the program that prints an invoice so that it instead constructs an object of type `Invoice`, using the classes of Section 22.7. Then invoke the `print` function of the `Invoice` object.