Web\_Database\_App\_PHP\_MySQL\_2E\_Hugh\_c05

apter 5. SQL and MySQL

In this chapter, we introduce the SQL database query language and the MySQL©[1] database

management system. Using our case study winestore database as a worked example, we show

you how to use SQL to define, manipulate, and query databases. At the end of this chapter,

you'll have the database skills to build a database tier for your web database applications.

[1] MySQL is a trademark of MySQL AB.

In this chapter, we cover the following topics:

• A short introduction to relational databases

• A quick start guide to the example winestore database and its entity-relationship model

• The MySQL command interpreter and the basic features of MySQL

• Using SQL to create and drop databases and tables

• Using SQL to insert, delete, and update data

• Querying with SQL, illustrated through examples and a case study

We assume that you have already installed MySQL and loaded the sample winestore database.

If not, the guides in Appendix A through Appendix C will help you.

The techniques that we discuss are used to interact with MySQL after a database has been

designed and expressed as SQL statements. An introduction to relational modeling and design

can be found in Appendix E. Managing and using the MySQL database server, and more

advanced SQL features, are discussed in Chapter 15. Chapter 8 covers issues that arise when

multiple users are writing to web databases.

5.1. Database BasicsThe field of databases has its own terminology. Terms such as database, table, attribute, row,

primary key, and relational model have specific meanings and are used throughout this chapter.

In this section, we present an example of a simple database to introduce the basic components

of relational databases, and we list and define selected terms used in the chapter. We then

show you our winestore database that we use throughout our examples in this chapter, and as

the basis of our sample application in Chapter 16 through Chapter 20. More detail on the

database can be found in Appendix E.

5.1.1. Introducing Relational Databases

A simple example relational database is shown in Figure 5-1. This database stores data about

wineries and the wine regions they are located in. A relational database is organized into tables,

and there are two tables in this example: a winery table that stores information about wineries,

and a region table that has information about wine regions. Tables collect together information

that is about one object.

Figure 5-1. An example relational database containing two related tables

Databases are managed by a database management system (DBMS) or database server. A

database server supports a database language to create and delete databases and to manage

and search data. The database language used by almost all database servers is SQL, a set of

statements that define and manipulate data. After creating a database, the most common SQL

statements used are INSERT, UPDATE, DELETE, and SELECT, which add, change, remove,

and search data in a database, respectively.

In this book, we use the MySQL database server to manage databases. MySQL runs as a

server (daemon) process or service, like Apache or IIS, and supports several different clients

including a command-line interpreter (that we use in this chapter) and a PHP function library(that we use throughout later chapters). One MySQL server can manage multiple databases for

you for multiple applications, and each can store different data organized in different ways.

A database table may have multiple attributes, each of which has a name. For example, the

winery table in Figure 5-1 has four attributes, winery ID, winery name, address, and region ID.

A table contains the data as rows, and a row contains values for each attribute that together

represent one related object. (Attributes are also known as fields or columns, while rows are

also known as records. We use attribute and row throughout this book.)

Consider an example. The winery table has five rows, one for each winery, and each row has a

value for each attribute. For example, in the first winery row, the attribute winery ID has a value

of 1, the winery name attribute has a value of Moss Brothers, the attribute address has a value

of Smith Rd., and the region ID attribute has a value of 3. There is a row for region 3 in the

region table and it corresponds to Margaret River in Western Australia. Together this data forms

the information about an object, the Moss Brothers Winery in Western Australia.

In our example, the relationship between wineries and regions is maintained by assigning a

region ID to each winery row. The region ID value for each region is unique, and this allows you

to unambiguously discover which region each winery is located in. Managing relationships using

unique values is fundamental to relational databases. Indeed, good database design requires

that you can make the right choice of which objects are represented as tables and which

relationships exist between the tables. We discuss good database design in Appendix E.

In our example of the relationship between wineries and regions, there's a one-to-many

mapping between regions and wineries: more than one winery can be situated in a region (three

wineries in the example are situated in the Barossa Valley) but a winery can be situated in only

one region. It's also possible to have two other types of relationship between tables: a one-to-

one relationship where, for example, each bottle of wine has one label design, and a many-to-

many relationship where, for example, many wines are delivered by many couriers. As we show

you later, unique values or primary keys allow these relationships to be managed and they're

essential to relational databases.

Attributes have data types. For example, in the winery table, the winery ID is an integer, the

winery name and address are strings, and the region ID is an integer. Data types are assigned

when a database is designed.

Tables usually have a primary key, which is formed by one or more values that uniquely identify

each row in a table. The primary key of the winery table is the winery ID, and the primary key of

the region table is the region ID. The values of these attributes aren't usually meaningful to the

user, they're just unique ordinal numbers that are used to uniquely identify a row of data and to

maintain relationships.

Figure 5-2 shows our example database modeled using entity-relationship (ER) modeling. An

ER model is a standard method for visualizing a database and for understanding the

relationships between the tables. It's particularly useful for more complex databases where

relationships of different types exist and you need to understand how to keep these up-to-date

and use them in querying. As we show you later, our winestore database needs a moderately

complex ER model.

In the ER model in Figure 5-2, the winery and region tables or entities are shown as rectangles.

An entity is often a real-world object and each one has attributes, where those that are part of

the primary key are shown underlined. The relationship between the tables is shown as a

diamond that connects the two tables, and in this example the relationship is annotated with anM at the winery-end of the relationship. The M indicates that there are potentially many winery

rows associated with each region. Because the relationship isn't annotated at the other end, this

means that there is only one region associated with each winery. We discuss ER modeling in

more detail in Appendix E.

Figure 5-2. An example relational model of the winery database

5.1.2. Database Terminology

Database

A repository to store data. For example, a database might store all of the data associated with

finance in a large company, information about your CD and DVD collection, or the records of an

online store.

Table

A part of a database that stores data related to an object, thing, or activity. For example, a table

might store data about customers. A table has columns, fields, or attributes. The data is stored

as rows or records.

Attributes

The columns in a table. All rows in a table have the same attributes. For example, a customer

table might have the attributes name, address, and city. Each attribute has a data type such as

string, integer, or date.

Rows

The data entries stored in a table. Rows contain values for each attribute. For example, a row in

a customer table might contain the values "Matthew Richardson," "Punt Road," and

"Richmond." Rows are also known as records.

Relational model

A formal model that uses database, tables, and attributes to store data and manages the

relationship between tables.(Relational) database management system (DBMS)

A software application that manages data in a database and is based on the relational model.

Also known as a database server.

SQL

A standard query language that interacts with a database server. SQL is a set of statements to

manage databases, tables, and data. Despite popular belief, SQL does not stand for Structured

Query Language and isn't pronounced Sequel: it's pronounced as the three-letter acronym S-Q-

L and it doesn't stand for anything.

Constraints

Restrictions or limitations on tables and attributes. A database typically has many constraints:

for example, a wine can be produced only by one winery, an order can't exist if it isn't

associated with a customer, and having a name attribute is mandatory for a customer.

Primary key

One or more attributes that contain values that uniquely identify each row. For example, a

customer table might have the primary key named cust ID. The cust ID attribute is then

assigned a unique value for each customer. A primary key is a constraint of most tables.

Index

A data structure used for fast access to rows in a table. An index is usually built for the primary

key of each table and can then be used to quickly find a particular row. Indexes are also defined

and built for other attributes when those attributes are frequently used in queries.

Entity-relationship (ER) modeling

A technique used to describe the real-world data in terms of entities, attributes, and

relationships. This is discussed in Appendix E.

Normalized database

A correctly designed database that is created from an ER model. There are different types or

levels of normalization, and a third-normal form database is generally regarded as being an

acceptably designed relational database. We discuss normalization in Appendix E.

5.1.3. The Winestore Database

This section is a summary of the entity-relationship model of the winestore database. It's

included for easy reference, and you'll find it useful to have at hand as you work through this

chapter.5.1.3.1. The winestore entity-relationship model

Figure 5-3 shows the complete entity-relationship model for our example winestore database;

this model is derived from the system requirements listed in Chapter 16, and is derived following

the process described in Appendix E. Appendix E also includes a description of the meaning of

each shape and line type used in the figure.

Figure 5-3. The winestore ER model

The winestore model can be summarized as follows:

• A customer at the online winestore purchases wines by placing one or more orders.

• Each customer has exactly one set of user details.

• Each customer has a title (such as "Mr" or "Dr") and lives in a country.

• Each order contains one or more items.

• Each item is a specific quantity of wine at a specific price.

• A wine is of a type such as "Red," "White," or "Sparkling."

• A wine has a vintage year; if the same wine has two or more vintages from different years,

these are treated as two or more distinct wines.

• Each wine is made by one winery.

• Each winery is located in one region.

• Each wine has one or more grape\_variety entries. For example, a wine of wine\_name"Archibald" might be made of the grape\_variety entries "Sauvignon" and "Cabernet." The

order of the entries is important. For example, a "Cabernet Sauvignon" is different from a

"Sauvignon Cabernet."

• Each inventory for a wine represents the on-hand stock of a wine. If a wine is available at two

prices, there are two inventories. Similarly, if the stock arrived at the warehouse at two

different times, there are two inventories.

• Each wine may have one or more inventories.

5.2. MySQL Command Interpreter

The MySQL command interpreter is commonly used to create databases and tables in web

database applications and to test queries. Throughout the remainder of this chapter we discuss

the SQL statements for managing a database. All these statements can be directly entered into

the command interpreter and executed. In later chapters, we'll show how to include SQL

statements in PHP scripts so that web applications can get and change data in a database.

Once the MySQL server is running, the command interpreter can be used. The command

interpreter can be run using the following command from the shell in a Unix or Mac OS X

system, assuming you've created a user hugh with a password shhh:

% /usr/local/bin/mysql -uhugh -pshhh

The shell prompt is represented here as a percentage character, %.

On a Microsoft Windows platform, you can access the command interpreter by clicking on the

Start menu, then the Run option, and typing into the dialog box:

"C:\Program Files\EasyPHP1-7\mysql\bin\mysql.exe" -uhugh -pshhh

Then, press the Enter key or click OK.

(For both Unix and Microsoft Windows environments, we're assuming you've installed MySQL in

the default directory location using our instructions in Appendix A through Appendix C.)

Running the command interpreter displays the output:

Welcome to the MySQL monitor. Commands end with ; or \g.

Your MySQL connection id is 3 to server version: 4.0.15-log

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

mysql>

The command interpreter displays a mysql> prompt and, after executing any command or

statement, it redisplays the prompt. For example, you might issue the statement:

mysql> SELECT NOW( );

This statement reports the time and date in the following output:

+---------------------+

| NOW( )

|+---------------------+

| 2004-03-01 13:48:07 |

+---------------------+

1 row in set (0.00 sec)

mysql>

After running a statement, the interpreter redisplays the mysql> prompt. We discuss the

SELECT statement later in this chapter.

As with all other SQL statements, the SELECT statement ends in a semicolon. Almost all SQL

command interpreters permit any amount of whitespace (spaces, tabs, or carriage returns) in

SQL statements, and they check syntax and execute statements only after encountering a

semicolon that is followed by a press of the Enter key.

We have used uppercase for the SQL statements throughout this book so that it's clear what's

an SQL statement and what isn't. However, any mix of upper- and lowercase is equivalent in

SQL keywords. Be careful, though: other parts of SQL statements such as database and table

names are case sensitive. You also need to be careful with values: for example, Smith, SMITH,

and smith are all different.

On startup, the command interpreter encourages the use of the help command. Typing help

produces a list of commands that are native to the MySQL interpreter and that aren't part of

SQL. All non-SQL commands can be entered without the terminating semicolon, but the

semicolon can be included without causing an error.

The MySQL command interpreter provides a lot of flexibility and many shortcuts:

• To quit the interpreter, type quit.

• The up- and down-arrow keys allow you to browse previously entered commands and

statements. On most platforms, the history of commands and statements is kept when you

quit the interpreter. When you run it again, you can once again scroll up using the up arrow

and execute commands and statements that were entered in the previous session.

• The interpreter has command completion. If you type the first few characters of a string that

has previously been entered and press the Tab key, the interpreter automatically completes

the command. For example, if wines is typed and the Tab key pressed, the command

interpreter outputs winestore, assuming the word winestore has been previously used.

• If there's more than one option that begins with the characters entered, or you wish the strings

that match the characters to be displayed, press the Tab key twice to show all matches.

You can then enter additional characters to remove any ambiguity and press the Tab key

again for command completion.

• If you're a Unix user, you can use a text editor to create SQL statements by entering the

command edit in the interpreter. This invokes the editor defined by the EDITOR shell

environment variable. After you exit the editor, the MySQL command interpreter reads,

parses, and runs the file created in the editor.

• You can run single commands and SQL statements without waiting for a MySQL command

prompt. This is particularly useful for adding SQL statements to startup scripts. For

example, to run SELECT now( ) from a Unix shell, enter the following command:% /usr/local/mysql/bin/mysql -uhugh -pshhh -e "SELECT now( );"

• You can create MySQL statements in a file using a text editor, and then load and run them.

For example, if you have statements stored in the file statements.sql, type the following into

the command interpreter to load and run the statements:

mysql> source statements.sql

You can also include a directory path before the filename. This feature is discussed in more

detail in Chapter 15.

•

Sometimes, you'll find you've mistyped a statement, forgotten a semicolon, or forgotten a

quote character. In most cases, to solve the problem you can type a semicolon and press

Enter: this causes MySQL to report an error and you can then start again. If you're missing

a matching quote character, type it in, then a semicolon, and then press Enter. If you're in a

real mess, type Control-C (by holding the Ctrl key and pressing C): this aborts the

command interpreter completely.

5.3. Managing Databases and Tables

In this section, we use the MySQL command interpreter to create databases and tables using

the winestore database as a case study. We also show you the statements that remove

databases and tables.

A discussion of advanced features is in Chapter 15. We show you how to manage indexes and

alter tables after they've been created, and delete and update data using queries and multiple

tables. We also show you how the details of how to store multiple statements in a file and

execute them; this is how we created our winestore script that you used in the installation steps

in Appendix A through Appendix C.

5.3.1. Creating Databases

The CREATE DATABASE statement creates a new, empty database without any tables or data.

The following statement creates a database called winestore:

mysql> CREATE DATABASE winestore;

A database name can be 64 characters in length at most and can contain any character except

the forward slash, backward slash, or period characters.

Database and table names are used as the disk file names that store the data. Therefore, if

your operating system has case-sensitive filenames, MySQL is case-sensitive to database and

table names; in general, Unix platforms are case sensitive and Microsoft Windows platforms

aren't. Attribute names are not case sensitive on all platforms. Aliases (which are discussed in

Chapter 15) are partially case sensitive: table aliases follow the same rule as table names (and

so are case sensitive on some platforms), while attribute aliases are case insensitive.

For the rest of this chapter, we omit the mysql> prompt from the command examples. To work

with a database, the command interpreter requires the user to be using a database before SQL

statements can be issued. Database servers have different methods for using a database and

these aren't part of the SQL standard. In the MySQL interpreter, you issue the command:

use winestore;5.3.2. Creating Tables

After issuing the use winestore command, you then usually enter statements to create the

tables in the database. Let's look one table from the winestore database, the customer table.

The statement that creates this table is shown in Example 5-1.

Example 5-1. Creating the customer table with SQL

The CREATE TABLE statement has three parts:

• Following the CREATE TABLE statement is a table name, which in this case is customer.

• Following an opening bracket is a list of attribute names, types and lengths, and modifiers.

These are comma separated.

• After this is a list of other information about the structure and use of the table. In this example,

a PRIMARY KEY is defined and the table type is set to MyISAM.

• Like all SQL statements, this one ends with a semi-colon.

We explain most of these in detail later in this section. Tables types are discussed in Chapter

15.

The CREATE TABLE statement for the customer table is derived from the entity-relationship

model in Figure 5-3, and the process of converting this model to CREATE TABLE statements is

described in Appendix E. The complete list of tables in the winestore database and a brief

description of each and its relationships is shown in Table 5-1.

Table 5-1. The tables in the winestore database

Table

Description

countries

Lookup table containing country names. Related to customer.

customer

Customer details, including address, contact details, and date of

birth. Related to countries, orders, titles, and users.grape\_variety Lookup table containing grape variety names. Related to

wine\_variety.

inventory

Stock records that show much wine is available and its price.

Related to wine.

items

The wines in an order and their quantity and price. Related to

wine and orders.

orders

Orders placed by customer, which contain items. Related to

customer and items.

region

Wine growing districts that contain wineries. Related to winery.

titles

users

wine

wine\_type

wine\_variety

winery

Lookup table containing titles (such as Mr. or Miss). Related to

customer.

Email addresses (which are also used as user names) and

encrypted passwords for each customer. Related to customer.

Details about the wines. Related to items, inventory, wine\_type,

wine\_variety, and winery.

Lookup table containing wine categories (such as red or white).

Related to wine.

The link between a wine and its grape varieties. Related to wine

and grape\_variety.

Winery details. Related to wine and region.

If you followed our installation instructions in Appendix A through Appendix C, you've already

downloaded the installation script that contains the statements to create all of the winestore

database tables and this has been loaded into your MySQL installation (along with example

data). To view the CREATE TABLE statements for the other tables in database, you can use the

SHOW CREATE TABLE command in the command interpreter. For example, to see the

statement used to create the wine table, type:

SHOW CREATE TABLE wine;

This statement is discussed in more detail in Chapter 15. You can also view the CREATE

TABLE statements by opening the installation file winestore.data in a text editor; this is a good

way to view all of the statements at once.

5.3.2.1. Tables and attributes

A table name can be 64 characters in length at most and may contain any character except a

forward slash or a period. As you've seen, the name is usually the name of an entity created in

the ER model. Attribute names may be up to 64 characters in length and can contain any

character.

There are many possible data types for attributes, and details of selected commonly-used types

are shown in Table 5-2. A complete list is provided in Section 6.2 of the MySQL manual. The

MySQL manual is found at http://www.mysql.com/documentation. You can also download a

copy from the same location and open it as a local file using your web browser; we recommend

this approach, as it allows you fast access to the manual.Table 5-2. Common SQL data types for attributes

Data type

Comments

int(length)

Integer with a maximum length; used for IDs,

age, counters, etc.

decimal(width[,decimal\_digits]) A number with a width including an optional

number of decimal\_digits after the decimal

point; used for currency, measurements, etc.

datetime

Stores a date and time in the format YYYY-MM-

DD HH:MM:SS.

time

Stores a time in the format HH:MM:SS.

dateStores a date in the format YYYY-MM-DD.

timestampStores the date and time in the format

YYYYMMDDHHMMSS.

The first-occurring timestamp attribute in a row

has a special property: it is set to the current

date and time when the row that contains it is

created and it updates each time the row that

contains it is modified. You can also update it to

the current date and time by setting the

attribute to NULL.

char(length)Any other timestamp attributes in a row do not

have this special property, but they can be

updated to the current date and time by

assigning NULL.

An unpadded, variable-length text string with a

specified maximum length.

A padded, fixed-length text string of size length.

blobAn attribute that stores up to 64 KB of data.

varchar(length)

For situations where the data stored is always much smaller or larger than the usual maximum

possible value, most attribute types can be defined as tiny, small, medium, and big. For

example, int can be specified as tinyint, smallint, mediumint, and bigint that are for signed

integers in the ranges -128 to 127, -32768 to 32767, -8388608 to 8388607, and

-9223372036854775808 to 9223372036854775807 respectively. The normal-size int has the

range -2147483648 to 2147483647. We recommend choosing the smallest type that is suitable

for a task: this saves space, and makes data retrieval and updates faster.

You'll find more detail of attribute types in Section 6.4 of the MySQL manual.

5.3.2.2. Modifiers

Modifiers may be applied to attributes. The most common modifier is NOT NULL, which means

that a row can't exist without this attribute having a value. For example:cust\_id int(5) NOT NULL,

Another common modifier is DEFAULT, which sets the data to the value that follows when no

data is supplied. For example, suppose you want to set the state attribute to the value Unknown

when it isn't provided. You can do this using:

state varchar(20) DEFAULT "Unknown",

DEFAULT and NOT NULL can be used in combination: if a value isn't supplied for an attribute,

NULL can be avoided by using the DEFAULT value; we return to this later in Section 5.4.

All numeric attributes have optional zerofill and unsigned modifiers. The former left-pads a value

with zeros up to the size of the attribute type. The latter allows only positive values to be stored

and roughly doubles the maximum positive value that can be stored.

Finally, the useful auto\_increment modifier is described in Section 5.4.

5.3.2.3. Keys

A primary key is one or more attributes that uniquely identify a row in a table. As we discussed

previously, primary keys are essential to maintaining relationships between tables in the

database, and every table should have one. In the customer table in Example 5-1, the primary

key is the cust\_id attribute: each customer has a unique cust\_id, and these are assigned

sequentially as customers are added to the table.

You don't always have to create an extra attribute that serves the purpose of being the primary

key. For example, in our users table we could choose the user\_name attribute as the primary

key, because each customer must have a unique email address. In our customer table, we

could also have defined the primary key to be the combination of the surname plus the

firstname plus the initial plus the zipcode (in the hope that's enough information to uniquely

identify a customer!). As this example illustrates, if you don't already have an attribute that

unique, it's easier to add an extra attribute that's purpose is to be the primary key. Determining

primary keys from an ER model is discussed in detail in Appendix E.

The final component of the CREATE TABLE statement includes a specification of the keys. In

Example 5-1, we specify that the unique identifier is the cust\_id attribute by adding the

statement PRIMARY KEY (cust\_id). The PRIMARY KEY constraint has two restrictions: the

attribute must be defined as NOT NULL, and any value inserted must be unique.

You can add other non-primary keys to a table. As we show you in Chapter 15, extra keys can

make querying and updating of data in the database much faster. Each additional key definition

creates an additional index that permits fast access to the data using the attributes defined in

the key. As an example, suppose you want to access the customer data by a surname and

firstname combination. In this case, you can add a KEY definition to the end of the CREATE

TABLE statement:

PRIMARY KEY (cust\_id),

KEY names (surname,firstname)

) type=MyISAM;

Each new KEY is given a unique label that you choose, in this case we've chosen the label

names.In many cases, without yet knowing what kinds of queries will be made on the database, it is

difficult to determine what keys you should specify. MySQL permits at least 16 indexes to be

created on any table (this depends on the table type), but unnecessary indexes should be

avoided. Each index takes additional storage space, and it must be updated by the database

server as the data stored in the table is inserted, deleted, and modified. In addition, indexes on

multiple attributes can only be used to speed up certain queries. We discuss how to use

indexes and index tuning in Chapter 15.

5.3.3. Deleting Databases and Tables

The DROP statement is used to remove tables and databases. Removing a table or database

also deletes the data contained in it. For example, to remove the customer table and its data,

use:

DROP TABLE customer;

To remove the complete winestore database (including all tables, indexes, and data), use:

DROP DATABASE winestore;

Take care with DROP—the command interpreter won't ask you if you're sure. However, we

show you how to prevent accidental deletion (and prevent other database users from deleting

databases, tables, and data) in Chapter 15.

Both DROP TABLE and DROP DATABASE support an optional IF EXISTS keyword which can

be used to prevent an error being reported if the database or table doesn't exist. For example,

to drop the winestore database and avoid an error if it's already been dropped (or was never

created), use:

DROP DATABASE IF EXISTS winestore;

We've used this feature at the beginning of the winestore.data file that contains the SQL

statements for loading the winestore database. The first three lines remove the database if it

exists, create a new database, and use the new database:

DROP DATABASE IF EXISTS winestore;

CREATE DATABASE winestore;

USE winestore;

You can therefore reload the file by following our instructions in Appendix A through Appendix C,

and it'll create and load a new winestore database every time.

5.4. Inserting, Updating, and Deleting Data

There are four major statements for working with data in SQL: SELECT, INSERT, DELETE, and

UPDATE. We describe the latter three statements in this section. SELECT is covered it in its

own section later in this chapter.

5.4.1. Inserting Data

Having created a database and the accompanying tables and indexes, the next step is to insertdata into the tables. Inserting a row can follow two different approaches. We show both

approaches by inserting the same data for a new customer, Lucy Williams.

Consider an example of the first approach using the customer table:

INSERT INTO customer VALUES (1,'Williams','Lucy','E',3,

'272 Station St','Carlton North','VIC','3054',12,'(613)83008460',

'2002-07-02');

The statement creates a new row in the customer table, then the first value 1 is inserted into the

first attribute, cust\_id. The second value 'Williams' is inserted into the second attribute surname,

'Lucy' into firstname, and so on.

The number of values inserted is the same as the number of attributes in the table (and an error

is generated if the number of values doesn't match the number of attributes). If you don't want to

supply data for an attribute, you can include NULL instead of a value (as long as the attribute

isn't defined as NOT NULL and NULL is valid for that data type). For example, to create a partial

customer row, you could use:

INSERT INTO customer VALUES (1,'Williams','Lucy',NULL,3,

NULL,NULL,NULL,NULL,12,NULL,NULL);

To create an INSERT statement using this first format, you need to know the ordering of the

attributes in the table. You can discover the table structure by typing SHOW COLUMNS FROM

customer into the MySQL command interpreter or by reviewing the CREATE TABLE statement

used to create the table. The SHOW statement is described in detail in Chapter 15.

If you want to insert more than one row, you can write more than one INSERT statement.

Alternatively, you can write one INSERT statement and separate each row with a comma.

Consider an example that uses the latter approach and inserts the details for two customers:

INSERT INTO customer VALUES (1,'Williams','Lucy','E',3,

'272 Station St','Carlton North','VIC','3054',12,'(613)83008460',

'2002-07-02'), (2,'Williams','Selina','J',4,'12 Hotham St',

'Collingwood','VIC','3066',12,'(613)99255432','1980-06-03');

This approach is the fastest way to insert data into MySQL.

Data can also be inserted using a second approach. Consider this example:

INSERT INTO customer SET cust\_id = 1, surname = 'Williams',

firstname = 'Lucy', initial='E', title\_id=3,

address='272 Station St', city='Carlton North',

state='VIC', zipcode='3054', country\_id=12,

phone='(613)83008460', birth\_date='2002-07-10';

In this approach, the attribute name is listed, followed by the assignment operator (=) and then

the value to be assigned. This approach doesn't require the same number of values as

attributes, and it also allows arbitrary ordering of the attributes. This can save you lots of typing

when a row has many attributes but is sparsely populated with values. For example, to create a

partial customer row, you could use:

INSERT INTO customer SET cust\_id = 653, surname = 'Williams',firstname = 'Lucy', title\_id = 3, country\_id = 12;

The first approach can actually be varied to function in a similar way to the second by including

parenthesized attribute names before the VALUES keyword. For example, you can create an

incomplete customer row with:

INSERT INTO customer (cust\_id, surname, city)

VALUES (1, 'Williams','North Carlton');

When inserting data, non-numeric attributes must be enclosed in either single or double quotes.

If a string contains single quotation marks, the string can be enclosed in double quotation

marks. For example, consider the string "Steve O'Dwyer". Likewise, strings containing double

quotation marks can be enclosed in single quotation marks. An alternative approach is to

escape the quotation character by using a backslash character; for example, as in the string

`Steve O\'Dwyer'. Numeric values can also be enclosed in quotes but they aren't mandatory.

There are other ways to insert data in addition to those discussed here. For example, a popular

variation is to insert data from another table using a query or to insert data from a formatted text

file. These two approaches and other variants are discussed in Chapter 15.

5.4.1.1. Defaults

If you don't include the value for an attribute, it is set to the DEFAULT value if it's supplied in the

table definition or to NULL otherwise (if it is valid for the attribute to be NULL). If an attribute is

defined as being NOT NULL and does not have a DEFAULT value, the value that's set depends

on the attribute type; for example, integer attributes are set to 0 (which causes an

auto\_increment attribute to be populated with a new identifier, as discussed next) and strings to

the empty string. However, rather than worry about what happens, we recommend that you

define a DEFAULT value for any attribute that you don't always want to list in an INSERT

statement. Even if you want NULL to be inserted when nothing is provided, you can define it as

the DEFAULT.

Inserting NULL into a TIMESTAMP (or any date or time type) attribute stores the current date

and time. Inserting 0 into a TIMESTAMP attribute doesn't have the same effect as inserting

NULL, because 0 is a valid date and time combination.

5.4.1.2. Auto-increment

MySQL provides a non-standard SQL auto\_increment modifier that makes management of

primary keys easy; most other database servers provide a similar non-standard feature. The

goal of using auto\_increment is to make sure that each row in your table has a unique primary

key so that you can refer to it in other tables; as discussed previously, this is a common

requirement in databases.

The following is a simple table definition that uses the auto\_increment feature to create a unique

value for the primary key:

CREATE TABLE names (

id smallint(4) NOT NULL auto\_increment,

name varchar(20),

PRIMARY KEY (id)

);

You can insert data into this table by setting only the name attribute:INSERT INTO names SET name = "Bob";

In this example, the id is set to the next available identifier because the default value of an

integer attribute is 0 and this invokes the auto\_increment feature.

In general, when you insert NULL (or zero) as the next value for an attribute with the

auto\_increment modifier, the value that is stored is the maximum value + 1. For example, if

there are already 10 rows in the names table with id values of 1 to 10, inserting a row with

NULL as the id (or not providing an id and invoking the default behavior) creates a row with an

id value of 11.

The auto\_increment modifier is a useful feature when you want to insert data with a unique

primary key, but don't want to have to read the data first to determine the next available value to

use. As we show you later in Chapter 8, this also helps avoid concurrency problems (and,

therefore, the need for locking) when several users are using the same database. The

disadvantage is that it's a proprietary MySQL feature. However, we also show you how to

develop a generic approach to managing identifiers in Chapter 9 and we also show you how it's

done with PHP's PEAR DB.

Only one attribute in a table can have the auto\_increment modifier.

The result of an auto\_increment modifier can be checked with the MySQL-specific function

last\_insert\_id( ). For the previous example, you can check which id was created with the

statement:

SELECT last\_insert\_id( );

This statement reports:

+------------------+

| last\_insert\_id( ) |

+------------------+

|

11 |

+------------------+

1 row in set (0.04 sec)

You can see that the new row has id=11. To check an identifier value, the function should be

called immediately after inserting the new row.

5.4.2. Deleting Data

The DELETE statement removes data from tables. For example, the following deletes all data in

the customer table but doesn't remove the table:

DELETE FROM customer;

A DELETE statement with a WHERE clause can remove specific rows; WHERE clauses are

frequently used in querying, and they are explained later in Section 5.5. Consider a simple

example:

DELETE FROM customer WHERE cust\_id = 1;

This deletes the customer with a cust\_id value of 1. Consider another example:

DELETE FROM customer WHERE surname = 'Smith';This removes all rows for customers with a surname value of Smith.

5.4.3. Updating Data

Data can be updated using a similar syntax to the INSERT statement. Consider an example:

UPDATE customer SET state = upper(state);

This replaces the string values of all state attributes with the same string in uppercase. The

function upper( ) is one of many MySQL functions discussed in Chapter 15.

You can update more than one attribute in a statement. For example, to set both the state and

city to uppercase, use:

UPDATE customer SET state = upper(state), city = upper(city);

The UPDATE statement is also often used with the WHERE clause. For example:

UPDATE customer SET surname = 'Smith' WHERE cust\_id = 7;

This updates the surname attribute of customer #7. Consider a second example:

UPDATE customer SET zipcode = '3001' WHERE city = 'Melbourne';

This updates the zipcode of all rows with a city value Melbourne.

After an UPDATE is completed, MySQL returns the number of rows that were changed. If

MySQL finds that a value doesn't need to be changed (because it's already set to the value you

want to change it to), it isn't updated and isn't included in the count that's returned.

5.5. Querying with SQL SELECT

The SELECT statement is used to query and retrieve one or more rows from a database. We

introduce it in this section, and then show you the WHERE clause for selecting data that

matches a condition. The section concludes with an introduction to the more advanced features

of SELECT statements and a short case study.

5.5.1. Basic Querying

Consider an example SELECT statement:

SELECT surname, firstname FROM customer;

This outputs the values of the attributes surname and firstname from all rows in the customer

table. Assuming we previously inserted four rows when we created the winestore database, the

output from the MySQL command interpreter is:

+-----------+-----------+

| surname | firstname |

+-----------+-----------+

| Marzalla | Dimitria |

| LaTrobe | Anthony |

| Fong

| Nicholas |

| Stribling | James |

+-----------+-----------+4 rows in set (0.04 sec)

Any attributes of a table may be listed in a SELECT statement by separating them with a

comma. If all attributes are required, the shortcut of an asterisk character (\*) can be used.

Consider the statement:

SELECT \* FROM region;

This outputs all the data from the table region:

+-----------+---------------------+

| region\_id | region\_name

|

+-----------+---------------------+

|

1 | All

|

|

2 | Goulburn Valley |

|

3 | Rutherglen

|

|

4 | Coonawarra

|

|

5 | Upper Hunter Valley |

|

6 | Lower Hunter Valley |

|

7 | Barossa Valley

|

|

8 | Riverland

|

|

9 | Margaret River

|

|

10 | Swan Valley

|

+-----------+---------------------+

10 rows in set (0.01 sec)

SELECT statements can also output data that isn't from a database. Consider the following

example:

SELECT curtime( );

This example runs a function that displays the current time:

+-----------+

| curtime( ) |

+-----------+

| 08:41:50 |

+-----------+

1 row in set (0.02 sec)

The SELECT statement can even be used as a simple calculator, using the MySQL

mathematical functions described in Chapter 15:

SELECT pi( )\*(4\*4);

This outputs:

+------------+

| pi( )\*(4\*4) |

+------------+

| 50.265482 |

+------------+

1 row in set (0.01 sec)5.5.2. WHERE Clauses

A WHERE clause is used as part of most SELECT queries to limit the rows that are retrieved to

those that match a condition.

Consider this grape-growing region table containing the details of ten regions:

mysql> SELECT \* from region;

+-----------+---------------------+

| region\_id | region\_name

|

+-----------+---------------------+

|

1 | All

|

|

2 | Goulburn Valley |

|

3 | Rutherglen

|

|

4 | Coonawarra

|

|

5 | Upper Hunter Valley |

|

6 | Lower Hunter Valley |

|

7 | Barossa Valley

|

|

8 | Riverland

|

|

9 | Margaret River

|

|

10 | Swan Valley

|

+-----------+---------------------+

10 rows in set (0.09 sec)

To show only the first three regions, you can type:

SELECT \* FROM region WHERE region\_id <= 3;

This outputs all attributes for the first three rows:

+-----------+-----------------+

| region\_id | region\_name |

+-----------+-----------------+

|

1 | All

|

|

2 | Goulburn Valley |

|

3 | Rutherglen

|

+-----------+-----------------+

3 rows in set (0.03 sec)

You can combine the attribute and row restrictions and select only the region\_name attribute for

the first three regions:

mysql> SELECT region\_name FROM region WHERE region\_id <= 3;

+-----------------+

| region\_name |

+-----------------+

| All

|

| Goulburn Valley |

| Rutherglen

|

+-----------------+

3 rows in set (0.01 sec)

The SQL Boolean operators AND and OR have the same function as the PHP && and ||operators introduced in Chapter 2. These can be used to develop more complex WHERE

clauses (and these can be combined with the MySQL functions described in Chapter 15).

Consider an example query:

SELECT \* FROM customer WHERE surname='Marzalla' AND firstname='Dimitria';

This retrieves rows that match both criteria, that is, those customers with a surname Marzalla

and a firstname Dimitria. In this example, you need to be careful to type the strings 'Marzalla'

and 'Dimitria' using the correct case because string values are case sensitive.

Consider a more complex example:

SELECT cust\_id FROM customer

WHERE (surname='Marzalla' AND firstname LIKE 'M%') OR

birth\_date='1980-07-14';

This finds rows with either the surname Marzalla and a firstname beginning with M, or

customers who were born on 14 July 1980; the LIKE operator is discussed in more detail in

Chapter 15. The OR operator isn't exclusive, so a row can contain a birth date of 14 July 1980,

a surname of Marzalla, and a firstname beginning with M. This query, when run on the

winestore database, returns:

+---------+

| cust\_id |

+---------+

| 440 |

| 493 |

+---------+

2 rows in set (0.01 sec)

SELECT queries are often sophisticated and a long WHERE clause may include many AND

and OR operators. More complex examples of queries are shown later in this chapter. As

discussed previously, the WHERE clause is also a common component of UPDATE and

DELETE statements.

5.5.3. Sorting and Grouping Output

Listing attributes in the SELECT statement and using WHERE allows you to decide what rows

and columns in a table are returned from a query. However, you might also want to sort the data

after it's returned, or you might want to group it together beforehand so that you can count the

number of rows with different values, find a minimum or maximum value, or sum a numeric field.

This section shows you how to pre- and post-process your data.

5.5.3.1. ORDER BY

The ORDER BY clause sorts the data after the query has been evaluated. Consider an

example:

SELECT surname, firstname FROM customer

WHERE city = 'Portsea' and firstname = 'James' ORDER by surname;

This query finds all customers who live in Portsea and who have the first name James. It then

presents the results sorted alphabetically by ascending surname:

+-----------+-----------+| surname | firstname |

+-----------+-----------+

| Leramonth | James |

| Mockridge | James |

| Ritterman | James |

+-----------+-----------+

3 rows in set (0.00 sec)

Sorting can be on multiple attributes. For example:

SELECT surname, firstname, initial FROM customer

WHERE city = 'Coonawarra' OR city = 'Longwood'

ORDER BY surname, firstname, initial;

This presents a list of customers who live in Coonawarra or Longwood, sorted first by ascending

surname, then (for those customers with the same surname) by firstname, and (for those

customers with the same surname and first name), by initial. The output for the winestore

customer table is:

Code View: Scroll / Show All

+------------+-----------+---------+

| surname | firstname | initial |

+------------+-----------+---------+

| Archibald | Belinda | Q

|

| Chester | Marie | S

|

| Dalion | Marie | C

|

| Eggelston | Martin | E

|

| Florenini | Melinda | O

|

| Holdenson | Jasmine | F

|

| Mellaseca | Craig | Y

|

| Mockridge | Dimitria | I

|

| Morfooney | Chris | K

|

| Nancarral | Samantha | W

|

| Oaton

| Joel

|V

|

| Oaton

| Rochelle | F

|

| Patton | Joel

|Z

|

| Patton | Penelope | E

|

| Patton | Samantha |

|

| Rosenthal | Chris | A

|

| Tonkin | Michelle | Z

|

| Tonnibrook | Belinda | T

|

+------------+-----------+---------+

18 rows in set (0.00 sec)

By default, the ORDER BY clause sorts in ascending order, or ASC. To sort in reverse or

descending order, DESC can be used. Consider an example:

SELECT \* FROM customer WHERE city='Melbourne' ORDER BY surname DESC;

5.5.3.2. GROUP BYThe GROUP BY clause is different from ORDER BY because it doesn't sort the data for output.

Instead, it sorts the data early in the query process, for the purpose of grouping or aggregation.

Grouping data using a sort is the easiest way to discover properties such as maximums,

minimums, averages, and counts of values.

Consider an example:

SELECT city, COUNT(\*) FROM customer GROUP BY city;

This query first sorts the rows in the customer table by city and groups the rows with matching

values together. The output of the query consists of two columns. The first is a sorted list of

unique cities. The second shows, for each city, the COUNT of the number of customers who live

in that city. The number of rows that are output is equal to the number of different city values in

the customer table, and the effect of COUNT(\*) is to count the number of rows per group.

Here are the first few lines output by the query:

+--------------+----------+

| city

| COUNT(\*) |

+--------------+----------+

| Alexandra |

14 |

| Armidale |

7|

| Athlone

|

9|

| Bauple

|

6|

| Belmont

|

11 |

| Bentley

|

10 |

| Berala

|

9|

| Broadmeadows |

11 |

So, for example, there are 14 customers who live in Alexandra, that is, 14 rows in the customer

table are grouped together because they have a city value of Alexandra.

The GROUP BY clause can find different properties of the aggregated rows. Here's an example:

SELECT city, MIN(birth\_date) FROM customer GROUP BY city;

This query first groups the rows by city and then shows the oldest customer in each city. The

first few rows of the output are as follows:

+---------------+-----------------+

| city

| MIN(birth\_date) |

+---------------+-----------------+

| Alexandra | 1938-04-01

|

| Armidale

| 1943-04-04

|

| Athlone

| 1943-04-04

|

| Bauple

| 1922-11-26

|

There are several functions that can be used in aggregation with the GROUP BY clause. Fiveparticularly useful functions are:

AVG( )

Finds the average value of a numeric attribute in a set

MIN( )

Finds a minimum value of a string or numeric attribute in a set

MAX( )

Finds a maximum value of a string or numeric attribute in a set

SUM( )

Finds the sum total of a numeric attribute

COUNT( )

Counts the number of rows in a set

The SQL standard places a constraint on the GROUP BY clause that MySQL doesn't enforce. In

the standard, all attributes that are selected (those that are listed immediately after the

SELECT statement) must appear in the GROUP BY clause. Most examples in this chapter don't

meet this unnecessary constraint.

5.5.3.3. HAVING

The HAVING clause permits conditional aggregation of data into groups. For example, consider

the following query:

SELECT city, count(\*), min(birth\_date) FROM customer

GROUP BY city HAVING count(\*) > 10;

The query groups rows by city, but only for cities that have more than 10 resident customers.

For those groups, the city, count of customers, and earliest birth date of a customer in that city

is output. Cities with less than 10 customers are omitted from the result set. The first few rows

of the output are as follows:

+--------------+----------+-----------------+

| city

| count(\*) | min(birth\_date) |

+--------------+----------+-----------------+

| Alexandra |

14 | 1938-04-01

|

| Belmont

|

11 | 1938-04-01

|

| Broadmeadows |

11 | 1955-10-13

| Doveton

|

13 | 1943-04-04

|

| Eleker

|

11 | 1938-04-01

|

| Gray

|

12 | 1943-04-04

|

|The HAVING clause must contain an attribute or expression (such as a function or an alias)

from the SELECT clause; in this example, count(\*) is listed after the SELECT and is used in the

HAVING condition.

The HAVING clause should be used exclusively with the GROUP BY clause. It is slow and

should never be used instead of a WHERE clause. For example, don't do this:

SELECT cust\_id, surname FROM customer HAVING surname = "Leramonth";

Do this instead:

SELECT cust\_id FROM customer WHERE surname = "Leramonth";

5.5.3.4. Combining clauses

You can combine ORDER BY, GROUP BY, HAVING, and WHERE. When all four are used, they

must appear in the order WHERE, then GROUP BY, then HAVING, and then ORDER BY. This

is intuitive because the WHERE clause picks the rows from the table, then GROUP BY

organizes the rows into sets, then HAVING picks the sets that match a condition, and then the

data is sorted by the ORDER BY condition just before it's output.

Consider an example. Suppose we want to find the number of customers with the same name

who live in each city in the state of Victoria, where the same name is defined as the same first

name and surname. For example, this might determine that there are five John Smiths who live

in Inverloch and three Tuong Nguyens in Carlton. Here's the query:

SELECT city, surname, firstname, count(\*) FROM customer

WHERE state = 'VIC'

GROUP BY surname, firstname, city HAVING count(\*) >= 2

ORDER BY city;

The query first uses the WHERE clause to pick the rows of customers that live in the state of

Victoria. The rows are then grouped together into sets, where the grouping condition is that the

customer surname and firstname are the same. Then, only those sets that have more than one

customer with the same name are kept by the HAVING clause; this gets rid of unique names.

Last, the ORDER BY clause sorts the customers by their city, and the city, first name, surname,

and count of the number of customers is output. Here is the output from the winestore customer

table:

+--------------+-----------+-----------+----------+

| city

| surname | firstname | count(\*) |

+--------------+-----------+-----------+----------+

| Broadmeadows | Mellaseca | Anthony |

| Eleker

| Leramonth | Harry |

2|

| Kalimna

| Galti | Nicholas |

2|

| Lucknow

| Mellili | Derryn |

2|

| McLaren

| Chester | Betty |

2|

+--------------+-----------+-----------+----------+

5 rows in set (0.00 sec)

2|

The output shows, for example, that there are two Betty Chesters who live in McLaren city in the

state of Victoria.

The GROUP BY clause sorts before it groups the rows into sets. Therefore, you don't need touse ORDER BY if you want the data to be output in the sort order used by the GROUP BY. For

example, you don't need to do this:

SELECT \* FROM customer GROUP BY surname ORDER BY surname;

If you leave out the ORDER BY clause, you'll get the same output:

SELECT \* FROM customer GROUP BY surname;

However, in practice, it doesn't really matter: the MySQL query optimizer will ignore the ORDER

BY clause if it's unnecessary. We discuss the query optimizer in Chapter 15.

5.5.3.5. DISTINCT

Suppose we want to find out which different cities our customers live in. The following query

shows the cities for all of the customers:

SELECT city FROM customer;

The problem is that a city name appears more than once if more than one customer lives in that

city. What we really want is a list of unique cities that the customers live in.

The DISTINCT clause presents only one example of each identical row from a query. We can

use it to find out the unique cities the customers live in:

SELECT DISTINCT city FROM customer;

This shows one example of each different city in the customer table.

This example has exactly the same result as:

SELECT city FROM customer GROUP BY city;

The DISTINCT clause is often slow to run, much like the GROUP BY and HAVING clauses. We

discuss how indexes and query optimization can speed queries in Chapter 15.

5.5.4. Limiting Output in MySQL

The LIMIT operator is MySQL-specific and is used to control the size of the output. For

example, the following query returns only the first five rows from the customer table:

SELECT \* FROM customer LIMIT 5;

This saves query evaluation time and reduces the size of the result set that's buffered in

memory by MySQL. It's particularly useful in a web database application where one page of

results is presented from a large table.

You can also specify which row to begin at, and then how many rows you want:

SELECT \* FROM customer LIMIT 100,5;

This returns the 100th to 104th rows from the customer table.

Row numbering begins at row zero. For example, if you want the first five rows of the customer

table, use:

SELECT \* FROM customer LIMIT 0,5;The following statement produces five rows beginning with row two:

SELECT \* FROM customer LIMIT 1,5;

Be careful: forgetting to count from zero is a common mistake.

If you want all rows after a particular row, the second parameter can be set to -1:

SELECT \* FROM customer LIMIT 600,-1;

For the winestore customer table, this returns 50 rows with cust\_id values of 601 to 650.

The LIMIT operator is included at the end of an SQL statement, after the optional WHERE,

GROUP BY, HAVING, and ORDER BY clauses.

5.6. Join Queries

You'll often want to output data that's based on relationships between two or more tables. For

example, in the winestore database, you might want to know which customers have placed

orders, which customers live in Australia, or how many bottles of wine Lucy Williams has

bought. These are examples of join queries, queries that match rows between tables based

(usually) on primary key values. In SQL, a join query matches rows from two or more tables

based on a condition in a WHERE clause and outputs only those rows that meet the condition.

As part of the process of converting the winestore entity-relationship model to SQL statements,

we've included the attributes required in any practical join condition. To understand which tables

can be joined in the winestore database, and how the joins are processed, it's helpful to have a

copy of the ER model at hand as you work your way through this section.

5.6.1. Beware of the Cartesian Product

Suppose you want to find out the names of the wineries in the winestore database and, for each

winery, the name of the region that it's located in. To do this, you examine the ER model and

discover that the region and winery tables are related, and that they both contain attributes that

you need in the answer to your query. Specifically, you need to retrieve the winery\_name

attribute from the winery table and the region\_name attribute from the region table, and you

need to join the two tables together to find the result.

Consider this query, which we might intuitively, but wrongly, use to find all the wineries in a

region:

SELECT winery\_name, region\_name FROM winery, region;

This query produces (in part) the following results:

+-------------------------------+-------------+

| winery\_name

| region\_name |

+-------------------------------+-------------+

| Durham and Sons Premium Wines | Coonawarra |

| Durham Brook Group

| Coonawarra |

| Durham Creek

| Coonawarra |

| Durham Estates

| Coonawarra |

| Durham Hill Vineyard

| Coonawarra |The impression here is that, for example, Durham Creek winery is located in the Coonawarra

region. This might not be the case. Why? First, you can use the techniques covered so far in

this chapter to check which region the Durham Creek winery is located in:

SELECT region\_id FROM winery WHERE winery\_name='Durham Creek';

The result is:

+-----------+

| region\_id |

+-----------+

|

9|

+-----------+

1 row in set (0.01 sec)

Now, you can query the region table to find the name of the region using:

mysql> SELECT region\_name FROM region WHERE region\_id=9;

+----------------+

| region\_name |

+----------------+

| Margaret River |

+----------------+

1 row in set (0.00 sec)

So, Durham Creek winery isn't in Coonawarra at all!

What happened in the first attempt at a join query? The technical answer is that you just

evaluated a Cartesian product: you produced as output all the possible combinations of wineries

and regions, most of which don't make any sense. These odd results can be seen if you add an

ORDER BY clause to the original query:

SELECT winery\_name, region\_name FROM winery, region

ORDER BY winery\_name, region\_name;

Recall that the ORDER BY clause sorts the results after the query has been evaluated and that

it has no effect on which rows are returned from the query. Here is the first part of the output:

+---------------------------------+---------------------+

| winery\_name

| region\_name

|

+---------------------------------+---------------------+

| Anderson and Sons Premium Wines | All

|

| Anderson and Sons Premium Wines | Barossa Valley

|

| Anderson and Sons Premium Wines | Coonawarra

|

| Anderson and Sons Premium Wines | Goulburn Valley |

| Anderson and Sons Premium Wines | Lower Hunter Valley |

| Anderson and Sons Premium Wines | Margaret River

|

| Anderson and Sons Premium Wines | Riverland

|

| Anderson and Sons Premium Wines | Rutherglen

|

| Anderson and Sons Premium Wines | Swan Valley

|

| Anderson and Sons Premium Wines | Upper Hunter Valley |

The query produces all possible combinations of the 10 region names and 300 wineries in thesample database! In fact, the number of rows output is the total number of rows in the first table

multiplied by the total rows in the second table. In this case, the output is 10 x 300 = 3,000

rows.

5.6.2. Elementary Natural Joins

A cartesian product isn't the join we want. Instead, we want to limit the results to only the

sensible rows, where the winery is actually located in the region. To do this, you need to

understand how the relationship between the region and winery tables is maintained. If you

examine the ER model, you'll see that many wineries are located in a region.

In the database tables, the relationship between the winery and region tables is maintained

using the primary key of the region table, the attribute region\_id that's also an attribute in the

winery table. To understand this, consider the first three rows from the winery table:

mysql> SELECT \* FROM winery LIMIT 3;

+-----------+--------------------------+-----------+

| winery\_id | winery\_name

| region\_id |

+-----------+--------------------------+-----------+

|

1 | Hanshaw Estates Winery |

2|

|

2 | De Morton and Sons Wines |

5|

|

3 | Jones's Premium Wines |

3|

+-----------+--------------------------+-----------+

3 rows in set (0.04 sec)

The first winery has a region\_id of 2, the second a region\_id of 5, and the third a region\_id of 3.

Consider now the first five rows of the region table:

mysql> SELECT \* FROM region LIMIT 5;

+-----------+---------------------+

| region\_id | region\_name

|

+-----------+---------------------+

|

1 | All

|

|

2 | Goulburn Valley |

|

3 | Rutherglen

|

|

4 | Coonawarra

|

|

5 | Upper Hunter Valley |

+-----------+---------------------+

5 rows in set (0.04 sec)

If you match up each winery's region\_id value with a region's region\_id value, you can

determine the relationship and answer the query. For example, you can now see that the first

winery (Hanshaw Estates Winery) is located in region 2, the Goulburn Valley.

From a querying perspective, we want to output winery\_name and region\_name values where

the region\_id in the winery table matches the corresponding region\_id in the region table. This

is a natural join.

You can perform a natural join on the winery and region tables using:

SELECT winery\_name, region\_name FROM winery NATURAL JOIN region

ORDER BY winery\_name;

The query produces (in part) the following sensible results:+---------------------------------+---------------------+

| winery\_name

| region\_name

|

+---------------------------------+---------------------+

| Anderson and Sons Premium Wines | Coonawarra

| Anderson and Sons Wines

| Coonawarra

|

| Anderson Brothers Group

| Rutherglen

|

| Anderson Creek Group

| Riverland

|

| Anderson Daze Group

| Rutherglen

|

| Anderson Daze Vineyard

| Margaret River

|

| Anderson Daze Wines

| Barossa Valley

|

| Anderson Ridge Wines

| Lower Hunter Valley |

|

A natural join query relies on the DBMS matching attributes with the same name across the two

tables. In this example, MySQL discovers that there's a region\_id attribute in the winery and

region tables, and it only outputs combinations where the region\_id in both tables is the same.

You can write a join query that explicitly specifies which attributes should be matched to

produce the correct result. The following query uses a WHERE clause to produce identical

results to our previous example:

SELECT winery\_name, region\_name FROM winery, region

WHERE winery.region\_id = region.region\_id

ORDER BY winery\_name;

We recommend writing out your joins so that they include the join condition in the WHERE

clause. This is safer and clearer than relying on the NATURAL JOIN operator to discover

common attribute names across tables and allowing the DBMS to figure out how the join is

done.

Several features are shown in this second example:

• The FROM clause contains the two table names winery and region, and so retrieves rows

from both tables.

• Attributes in the WHERE clause are specified using both the table name and attribute name,

separated by a period. This is useful because the same attribute name is often used in

different tables, and the query can't figure out which table is meant unless you include it.

When an attribute name occurs in only one table, you can omit the table name.

• In this example, region\_id in the region table and region\_id in the winery table have to be

specified unambiguously as region.region\_id and winery.region\_id. In contrast,

winery\_name and region\_name don't need the table name because they occur only in the

winery and region tables respectively.

• The use of both the table and attribute name can also be used for clarity in queries, even if it

isn't required. So, for example, you could write winery.winery\_name in the example query.

It can also be used in all parts of the query, not just the WHERE clause.

• The WHERE clause includes a join clause that matches rows between the multiple tables. In

this example, the output is reduced to those rows where wineries and regions have

matching region\_id attributes, resulting in a list of all wineries and which region they are

located in. This is the key to joining two or more tables to produce sensible results.5.6.2.1. Examples

A join can be used to find lots of useful information from the winestore database. Suppose we

want to find the names of wineries and the wines they make. Again, after examining the ER

model, you'll see that you need to join together the related wine and winery tables to get the

required names. Here's the query you'd need to write to get the correct result:

SELECT winery\_name, wine\_name FROM winery, wine

WHERE wine.winery\_id = winery.winery\_id;

This query joins the winery and wine tables by matching the winery\_id attributes. The result is

the names and wineries of the 1,048 wines stocked at the winestore.

You can extend this query to produce a list of wines made by a specific winery or group of

wineries. For example, to find all wines made by wineries with a name beginning with Borg, use:

SELECT winery\_name, wine\_name FROM winery, wine

WHERE wine.winery\_id = winery.winery\_id

AND winery.winery\_name LIKE 'Borg%';

The LIKE clause is discussed in detail in Chapter 15.

Here are two more example join queries:

• To find the name of the region that the Ryan Ridge Winery is situated in:

SELECT region\_name FROM region, winery

WHERE winery.region\_id=region.region\_id

AND winery\_name='Ryan Ridge Winery';

• To find which wineries make Tonnibrook wines:

SELECT winery\_name FROM winery, wine

WHERE wine.winery\_id=winery.winery\_id

AND wine\_name='Tonnibrook';

5.6.2.2. Using DISTINCT in joins

The next example uses the DISTINCT operator to find wines that cost less than $10:

SELECT DISTINCT wine.wine\_id FROM wine, inventory

WHERE wine.wine\_id=inventory.wine\_id AND cost<10;

Wines can have more than one inventory row, and the DISTINCT operator shows each wine\_id

once by removing any duplicates.

Here are two examples that use DISTINCT to show only one matching answer:

• To find which countries the customers live in:

SELECT DISTINCT country FROM customer, countries

WHERE customer.country\_id = countries.country\_id;

• To find which customers have ordered wines:

SELECT DISTINCT surname,firstname FROM customer,ordersWHERE customer.cust\_id = orders.cust\_id

ORDER BY surname,firstname;

5.6.3. Joins with More than Two Tables

Queries can join more than two tables. Suppose you want to find the details of the wine

purchases made by a customer, including the customer's details, the dates they made an order,

and the quantity and price of the items purchased. You examine the ER model, and see that the

customer table that contains the customer information is related to the orders table that contains

the date, and the orders table is related to the items table that contains the quantities and

prices. So, to get the information you need, you have to join all three tables together.

By examining the database structure or the CREATE TABLE statements, you can see that the

cust\_id attribute can be used to join together the customer and the orders table. Joining the

orders table and items table is a little trickier: the primary key of the orders table isn't just the

order\_id, it's both the cust\_id and the order\_id. So, for example there are many rows with an

order\_id of 1, but what makes a row unique is the combination of the cust\_id for a customer and

the order\_id. These two attributes together are used to join the orders and items tables.

Suppose now that we want run this query for customer #2. Here's the query you'd use:

SELECT \* FROM customer, orders, items

WHERE customer.cust\_id = orders.cust\_id AND

orders.order\_id = items.order\_id AND

orders.cust\_id = items.cust\_id AND customer.cust\_id = 2;

The WHERE clause contains the join condition between the three tables, customer, orders, and

items, and the rows selected are those in which the cust\_id is the same for all three tables, the

cust\_id is 2, and the order\_id is the same in the orders and items tables. The example illustrates

how frequently the Boolean operators AND and OR are used.

If you remove the cust\_id=2 clause, the query outputs all items from all orders by all customers.

This is a large result set, but still a sensible one that is much smaller than the cartesian product!

Here are two more examples that join three tables:

• To find which wines are made in the Margaret River region:

SELECT wine\_id FROM wine, winery, region

WHERE wine.winery\_id=winery.winery\_id AND

winery.region\_id=region.region\_id AND

region.region\_name='Margaret River';

• To find which region contains the winery that makes wine #28:

SELECT region\_name FROM wine, winery, region

WHERE wine.winery\_id=winery.winery\_id AND

winery.region\_id=region.region\_id AND

wine.wine\_id=28;

Extending to four or more tables generalizes the approach further. To find the details of

customers who have purchased wines from the Ryan Estates Group winery, use:

SELECT DISTINCT customer.cust\_id, surname, firstnameFROM customer, winery, wine, items

WHERE customer.cust\_id=items.cust\_id AND

items.wine\_id=wine.wine\_id AND

wine.winery\_id=winery.winery\_id AND

winery.winery\_name='Ryan Estates Group'

ORDER BY surname, firstname;

This query is the most complex so far and has four parts. The easiest way to understand a

query is usually to start at the end of the WHERE clause and work toward the SELECT clause:

1. The WHERE clause restricts the winery rows to the Ryan Estates Group (which, in this case,

only matches one winery).

2. The resultant winery row is joined with the wine table to find all wines made by the Ryan

Estates Group.

3. The wines made by Ryan Estates Group are joined with the items that have been purchased

by joining to the items table.

4. The purchased wines are then joined with the customer rows to find the purchasers. You can

leave out the orders table, because the items table contains a cust\_id for the join; if you

need the order number or credit card number (or another orders attribute), the orders

table needs to be included in the query.

5. The result is the details of customers who have purchased Ryan Estates Group wines. The

DISTINCT clause is used to show each customer only once. ORDER BY sorts the

customer rows into telephone directory order.

Designing a query like this is a step-by-step process. We began by testing a query to find the

winery\_id of wineries with the name Ryan Estates Group. Then, after testing the query and

checking the result, we progressively added additional tables to the FROM clause and the join

conditions. Finally, we added the ORDER BY clause.

The next example uses three tables. It queries the complex many-to-many relationship that

exists between the wines and grape\_variety tables via the wine\_variety table. A wine can have

one or more grape varieties and these are listed in a specific order (e.g., Cabernet, then

Sauvignon). From the other perspective, a grape variety such as Cabernet can be in hundreds

of different wines. The many-to-many relationship is managed by creating an intermediate table

between grape\_variety and wine called wine\_variety. The id attribute value stored in that table

represents the order in which the grape varieties should appear for the wine. You can find a

longer discussion of how these tables were designed and how they're used in Appendix E.

Here is the example query that joins the three tables to find what grape varieties are in wine

#1004:

SELECT variety FROM grape\_variety, wine\_variety, wine

WHERE wine.wine\_id=wine\_variety.wine\_id AND

wine\_variety.variety\_id=grape\_variety.variety\_id AND

wine.wine\_id=1004

ORDER BY wine\_variety.id;

The result of the query is:+-----------+

| variety |

+-----------+

| Cabernet |

| Sauvignon |

+-----------+

2 rows in set (0.00 sec)

The join condition is the same as any three-table query. The only significant difference is the

ORDER BY clause that presents the results in id order (the first listed variety was stored with

ID=1, the second ID=2, and so on).

5.7. Case Study: Adding a New Wine

In this section, we show you an example that combines some of the statements we've

discussed in this chapter, and shows you the basics of writing data to databases.

In this example, let's insert a new wine into the database using the MySQL command-line

interpreter. Let's suppose that 24 bottles of a new wine, a Curry Cabernet Merlot 1996 made by

Rowley Brook Winery, have arrived, and you wish to add a row to the database for the new

wine. This new wine costs $14.95 per bottle.

The addition has several steps, the first of which is to find out the next available wine\_id. You

need to do this because we're not using the MySQL-proprietary auto\_increment feature in the

winestore database. Here's the query:

SELECT max(wine\_id) FROM wine;

This reports:

+--------------+

| max(wine\_id) |

+--------------+

|

1048 |

+--------------+

1 row in set (0.00 sec)

Now, we can use an INSERT INTO statement to create the basic row for the wine in the wine

table:

INSERT INTO wine SET wine\_id=1049, wine\_name='Curry Hill', year=1996,

description='A beautiful mature wine. Ideal with red meat.';

This creates a new row and sets the basic attributes. The wine\_id is set to the 1048 + 1 = 1049.

The remaining attributes (the wine\_type identifier, the winery\_id identifier, and the varieties in

the wine\_variety table) require further querying and then subsequent updates.

The second step is to set the winery\_id for the new wine. We need to search for the Rowley

Brook Winery winery to identify the winery\_id:

SELECT winery\_id FROM winery WHERE winery\_name='Rowley Brook Winery';

The result returned is:+-----------+

| winery\_id |

+-----------+

|

298 |

+-----------+

1 row in set (0.00 sec)

We can now update the new wine row to set the winery\_id=298:

UPDATE wine SET winery\_id = 298 WHERE wine\_id = 1049;

The third step is similar to the second, and is to set the wine\_type identifier in the wine table.

You can discover the wine\_type\_id for a Red wine using:

SELECT wine\_type\_id FROM wine\_type WHERE wine\_type = "Red";

This reports that:

+--------------+

| wine\_type\_id |

+--------------+

|

6|

+--------------+

1 row in set (0.01 sec)

Now, you can set the identifier in the wine table:

UPDATE wine SET wine\_type = 6 WHERE wine\_id = 1049;

The fourth step is to set the variety information for the new wine. We need the variety\_id values

for Cabernet and Merlot. These can be found with a simple query:

SELECT \* FROM grape\_variety;

In part, the following results are produced:

+------------+------------+

| variety\_id | variety |

+------------+------------+

|

1 | Riesling |

|

2 | Chardonnay |

|

3 | Sauvignon |

|

4 | Blanc

|

|

5 | Semillon |

|

6 | Pinot

|

|

7 | Gris

|

|

8 | Verdelho |

|

9 | Grenache |

|

10 | Noir

|

|

11 | Cabernet |

|

12 | Shiraz |

|

13 | Merlot |Cabernet has variety\_id=11 and Merlot variety\_id=13. We can now insert two rows into the

wine\_variety table. Because Cabernet is the first variety, set its ID=1, and ID=2 for Merlot:

INSERT INTO wine\_variety SET wine\_id=1049, variety\_id=11, id=1;

INSERT INTO wine\_variety SET wine\_id=1049, variety\_id=13, id=2;

The final step is to insert the first inventory row into the inventory table for this wine. There are

24 bottles, with a per-bottle cost of $14.95:

INSERT INTO inventory SET wine\_id=1049, inventory\_id=1, on\_hand=24,

cost=14.95, date\_added="04/03/01";

We've finished inserting the wine into the database. Now, to conclude, let's retrieve the details

of the wine to make sure everything is as it should be. We'll retrieve the wine name, its year, the

winery, the varieties, the wine type, and its cost. Here's the query:

SELECT year, wine\_name, winery\_name, variety, wine\_type.wine\_type, cost

FROM wine, winery, wine\_variety, grape\_variety, wine\_type, inventory

WHERE wine.wine\_id = 1049 AND

wine.wine\_id = wine\_variety.wine\_id AND

wine\_variety.variety\_id = grape\_variety.variety\_id AND

wine.wine\_type = wine\_type.wine\_type\_id AND

wine.winery\_id = winery.winery\_id AND

wine.wine\_id = inventory.wine\_id

ORDER BY wine\_variety.id;

The WHERE clause looks complicated, but it just joins together all of the tables in the FROM

clause by matching up the identifier attributes and specifies we want for wine #1049. Here's the

output:

+------+------------+---------------------+----------+-----------+-------+

| year | wine\_name | winery\_name

| variety | wine\_type | cost |

+------+------------+---------------------+----------+-----------+-------+

| 1996 | Curry Hill | Rowley Brook Winery | Cabernet | Red

| 14.95 |

| 1996 | Curry Hill | Rowley Brook Winery | Merlot | Red

| 14.95 |

+------+------------+---------------------+----------+-----------+-------+

2 rows in set (0.01 sec)

Two rows are returned because there are two varieties for this wine in the wine\_variety table.

We've now covered as much complex querying in SQL as we need for you to develop most web

database applications. You'll find a discussion of advanced features you can use in Chapter 15.

Beginning in the next chapter, we show you how to include SQL statements in PHP scripts to

automate querying and build web database applications