SQL for MySQL

A Beginner's Tutorial

Djoni Darmawikarta

SQL for MySQL: A Beginner's Tutorial
Copyright © 2014 Brainy Software Inc.
First Edition: June 2014

ISBN: 978-0-9808396-7-8

Book and Cover Designer: Mona Setiadi

Technical Reviewer: Budi Kurniawan
Indexer: Chris Mayle

Warning and Disclaimer

Table of Contents

Introduction

Welcome to *SQL for MySQL: A Beginner's Tutorial*. This book is for you if you want to learn SQL the easy way. SQL, which stands for Structured Query Language and is pronounced es-cue-el, is the standard language you use to interact with a relational database management system (RDBMS). This book uses the free edition of the MySQL database to show how SQL works.

SQL Overview

Initially developed at IBM in the early 1970s, SQL was formalized by the American National Standards Institute (ANSI) in 1986. Since then the SQL standard has been revised seven times. The examples in this book were tested using MySQL 5.6, which conforms to the SQL:2008 standard. This standard is one revision earlier than SQL:2011, the latest standard.

SQL consists of a data definition language (DDL) and a data manipulation language (DML). The DDL is used to create, delete, and alter the structure of a table and other database objects. The DML is used to insert, retrieve, and update data in a table or tables.

Many database vendors implement a version of SQL that is not 100% compliant with the standard. They often add unique features to their SQL, resulting in an SQL dialect. For example, the following are some of the differences between MySQL and Oracle.

- The AS reserved word in the CREATE TABLE AS INSERT statement is optional in MySQL but mandatory in Oracle.
- A MySQL INSERT statement can insert multiple rows; an Oracle INSERT statement can only insert one row.

- MySQL only supports UNION set operator, whereas Oracle supports UNION, INTERSECT and MINUS set operators.
- MySQL’s stored routine equivalent in Oracle is its stored routine that you create using Oracle’s PL/SQL (Procedural Language extension to SQL). MySQL does not give a name to its procedural language extension that you use to create its stored routines. PL/SQL has much more features than for creating stored routines.

Because of these dialects, SQL statements written for one RDBMS may not necessarily work in other RDBMS’s.

About This Book

This book consists of thirteen chapters and three appendixes. This section gives you an overview of each chapter and appendix.

Chapter 1, “Storing and Maintaining Data” starts the book by discussing how data is stored and maintained in a relational database. In this chapter you learn how to use SQL INSERT, UPDATE, and DELETE statements.

Chapter 2, “Basic Queries” explains how to construct queries using the SELECT statement.

Chapter 3, “Full-Text Search” explores the distinct MySQL full-text search features.

Chapter 4, “Query Output” shows how you can format query outputs beyond simply displaying columns of data from a database.

Chapter 5, “Grouping” explains what a group is, how to create a group, and how to apply aggregate functions to a group.

Chapter 6, “Joins” talks about the JOIN clause for querying data from multiple tables.

Chapter 7, “Subqueries” discusses the subquery. A subquery is a query that can be nested in another query.

Chapter 8, “Compound Queries” talks about set operators for combining the outputs of multiple queries.

Chapter 9, “Views” discusses views, which are predefined queries that you create and store in a database.

Chapter 10, “Built-in Functions” discusses some of the most commonly used built-in functions in the MySQL database.

Chapter 11, “Transaction” shows how to treat multiple SQL statements as a transaction set.

Chapter 12, “Stored Routines” introduces MySQL’s stored routines. Stored routines extend SQL.

Chapter 13, “The Data Dictionary” shows how to use the data dictionary, the metadata of a database, to find information about the database.

Appendix A, “Installing MySQL Community Edition” is a guide to installing MySQL Database Community Edition and making preparations for trying out the book examples.

Appendix B, “MySQL Built-in Data Types” provides a list of MySQL built-in data types.

Finally, Appendix C, “Indexing” covers the various indexing techniques available in the MySQL database.

Code Conventions

SQL is not case sensitive. In this book, however, SQL reserved words such as CREATE and SELECT and keywords such as COUNT and MAX are written in upper case. Non-reserved words, such as table and column names, are written in lower case.

In the examples accompanying this book, a single space is used between words or expressions. Extra spaces are allowed and have no effect.

Code Download

The examples accompanying this book can be downloaded from this site.

http://books.brainysoftware.com/download

Chapter 1
Storing and Maintaining Data

Data in a relational database (such as MySQL) is stored in tables. A very simple sales database, for example, might have four tables to store data on products, customers, suppliers, and customer orders.

When you add a record of data into a table, the record is stored as a row of the table. A record has fields. A product record, for example, might have four fields: **product_code**, **name**, **price**, and **launch_date**. All records you store in the product table must have the same fields. Each of the fields is a column of the table.

This chapter shows you how to use SQL statements to store and maintain data. The main objective of this chapter is to give you a taste of working with SQL.

To test the book examples you need a working MySQL database. Appendix A, "Installing MySQL Community Edition" shows how you can install MySQL Community Edition and make it ready for use with the examples. This appendix also shows you how to use the *mysql* command-line tool to interactively execute your SQL statements. If you do not have a working MySQL, you should read this appendix first.

Creating and Selecting A Default Database

You need a database to store your data. The CREATE DATABASE command syntax is as follows.

```
CREATE DATABASE database_name;
```

For example, to create a database named **sales**, which you will use to test the book examples, execute the following statement.

```
CREATE DATABASE sales;
```

Before you can use a database, you need to select it using the USE command. A database selected this way becomes the default database for that session. To use the **sales** database as your default database, for example, issue the following command.

```
USE sales;
```

Creating a Table

Before you can store data in a database, you must first create a table for your data. You do this by using the CREATE TABLE statement.

The syntax for the CREATE TABLE statement is as follows.

```
CREATE TABLE database_name.table_name
   (column_1 data_type_1,
    column_2 data_type_2,
    ...
    [PRIMARY KEY (columns)]
);
```

To create a table in your default database, you do not need to qualify the table name with the database name. From now on, the book examples will use **sales** as the default database.

Listing 1.1 shows a CREATE TABLE statement for creating a **product** table with four columns.

Listing 1.1: Creating a product table with four columns

```
CREATE TABLE product
  (
    p_code    VARCHAR (6),
    p_name    VARCHAR (15),
    price     DECIMAL(4,2),
    launch_dt DATE,
    PRIMARY KEY (p_code)
  );
```

The four columns have three different data types. They are as follows.

- VARCHAR (*n*) – variable length string up to *n* characters.

- DECIMAL(*p*, *s*) – numeric with precision *p* and scale *s*. The price column, whose type is DECIMAL(4,2), can store numbers between -99.99 and +99.99.
- DATE – date

Note
Appendix B, "MySQL Built-in Data Types" provides a complete list of MySQL data types.

When creating a table, you should always add a primary key, even though a primary key is optional. A primary key is a column or a set of columns that uniquely identify every row in the table. In the CREATE TABLE statement in Listing 1.1, the **p_code** field will be made the primary key for the product table.

Also note that an SQL statement must be terminated with a semicolon (;)

Adding Data

Once you have a table, you can add data to it using the INSERT statement. The syntax for the INSERT statement is as follows

```
INSERT INTO table
   (column_1,
    column_2,
    ... )
VALUES (value_1,
        value_2,
    ... )
);
```

For example, Listing 1.2 shows an SQL statement that inserts a row into the **product** table.

Listing 1.2: Inserting a row into the product table

```
INSERT INTO product
  ( p_code, p_name, price, launch_dt)
  VALUES ( 1, 'Nail', 10.0, '2013-03-31');
```

After you execute the statement in Listing 1.2, your product table will have one row. You can query your table using this statement.

```
SELECT * FROM product;
```

The query result will be as follows.

```
+--------+--------+-------+------------+
| p_code | p_name | price | launch_dt  |
+--------+--------+-------+------------+
| 1      | Nail   | 10.00 | 2013-03-31 |
+--------+--------+-------+------------+
```

You can insert more than one row in an INSERT statement. The INSERT statement in Listing 1.3 add five more rows to the **product** table.

Listing 1.3: Adding five more rows to the product table

```
INSERT INTO product (p_code, p_name, price, launch_dt)
  VALUES (2, 'Washer', 15.00, '2013-03-29'),
 (3, 'Nut', 15.00, '2013-03-29'),
 (4, 'Screw', 25.00, '2013-03-30'),
 (5, 'Super_Nut', 30.00, '2013-03-30'),
 (6, 'New Nut', NULL, NULL);
```

After executing the statements in Listing 1.3, your product table will contain these rows.

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 1      | Nail      | 10.00 | 2013-03-31 |
| 2      | Washer    | 15.00 | 2013-03-29 |
| 3      | Nut       | 15.00 | 2013-03-29 |
| 4      | Screw     | 25.00 | 2013-03-30 |
| 5      | Super_Nut | 30.00 | 2013-03-30 |
| 6      | New Nut   |  NULL | NULL       |
+--------+-----------+-------+------------+
```

Updating Data

You use the UPDATE statement to update one or more columns of existing data. You can update all rows in a table or certain rows in the table.

The syntax for the UPDATE statement is as follows

```
UPDATE table_name
SET column_1 = new_value_1 [,
    column_2 = new_value_2,
    ... ]
[WHERE condition];
```

You specify which rows to update in the WHERE clause. Without a WHERE clause, all rows will be updated. With a WHERE clause, only rows that meet the condition will be updated. If no row meets the condition in the WHERE clause, nothing will be updated.

As an example, the SQL statement in Listing 1.4 cuts the price by 5%. As the UPDATE statement does not have a WHERE clause, the prices of all the products will be updated.

Before you execute Listing 1.4, issue a SET AUTOCOMMIT = 0; command.

Listing 1.4: Updating the price column

```
UPDATE product
SET price = price - (price * 0.05);
```

If you query the **product** table using this statement, you will learn that the values in the price column have changed.

```
SELECT * FROM product;
```

Here is the result of the query.

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 1      | Nail      | 9.50  | 2013-03-31 |
| 2      | Washer    | 14.25 | 2013-03-29 |
| 3      | Nut       | 14.25 | 2013-03-29 |
| 4      | Screw     | 23.75 | 2013-03-30 |
| 5      | Super_Nut | 28.50 | 2013-03-30 |
| 6      | New Nut   |  NULL | NULL       |
+--------+-----------+-------+------------+
```

Now, issue a ROLLBACK command to return the data values back to before the update:

```
ROLLBACK;
```

As another example, the statement in Listing 1.5 updates the price of product with **p_code** = 9. Since there is no row with such a p_code, no row will be updated.

Listing 1.5: Updating the price column with a WHERE clause

```
UPDATE product
SET price = price - (price * 0.05)
WHERE p_code = 9;
```

Deleting Data

To delete a row or multiple rows in a table, use the DELETE statement. You can specify which rows to be deleted by using the WHERE clause.

The syntax for the DELETE statement is as follows

```
DELETE FROM table
[WHERE condition];
```

You specify which rows to delete in the WHERE clause.

For example, the statement in Listing 1.6 deletes from the **product** table all rows whose **p_name** field value is ‘Nut’.

Listing 1.6: Deleting rows

```
DELETE FROM product
WHERE p_name = 'Nut';
```

After you run the statement in Listing 1. 6, please issue a ROLLBACK command to return the data values back to before the deletion:

```
ROLLBACK;
```

If none of the rows meets the condition, nothing will be deleted. Without the WHERE condition, all rows will be deleted and the **product** table will be empty.

As another example, the SQL statement in Listing 1.7 deletes all the rows in the **product** table. If you really execute this statement, please issue a ROLLBACK statement to get back all the rows before deletion.

Listing 1.7: Deleting all rows

```
DELETE FROM product;
```

Note that you cannot delete some of the columns in a row; the DELETE statement deletes the whole row. If you need to change the content of a specific column, use the UPDATE statement. For instance, the statement in Listing 1.8 changes the content of the price column to NULL. NULL is the absence of a value; it is neither 0 (zero) nor empty. Chapter 2, “Basic Queries” has a section (“Handling NULL”) that explains NULL in detail.

Listing 1.8: Updating to NULL

```
UPDATE product SET price = NULL WHERE p_name = 'Nut';
```

When you query the Nut product, the result will show NULL on the price column.

```
SELECT * FROM product WHERE p_name = 'Nut';
```

The output is as follows.

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 3      | Nut       | NULL  | 2013-03-29 |
+--------+-----------+-------+------------+
```

Please issue a ROLLBACK command to return the data values back to before the update:

```
ROLLBACK;
```

Summary

In this chapter you learned how to create a table and store data. In Chapter 2, “Basic Queries” you will learn to use the SELECT statement to query data.

Chapter 2
Basic Queries

A query is a request for data from one or more tables. When you execute a query, rows that satisfy the condition of the query will be returned as a table. Similarly, when a query embedded in another query or a program gets executed, the data returned to the other query or the program is a table.

In this chapter you learn how to write basic queries using the SELECT statement. Once you master basic queries, you can start learning about queries within other queries in Chapter 7, "Subqueries" and within stored routines in Chapter 12, "Stored Routines."

The SELECT statement

All queries regardless of their complexity use the SELECT statement. The SELECT statement has the following general syntax.

```
SELECT column_names FROM table [WHERE condition];
```

Only the SELECT and FROM clauses are mandatory. If your query does not have a WHERE clause, the result will include all rows in the table. If your query has a WHERE clause then only the rows satisfying the WHERE condition will be returned.

Querying All Data

The simplest query, which reads all data (all rows and all columns) from a table, has the following syntax.

```
SELECT * FROM table;
```

The asterisk (*) means all columns in the table. For instance, Listing 2.1 shows an SQL statement that queries all data from the **product** table.

Listing 2.1: Querying all product data

```
SELECT * FROM product;
```

Executing the query will give you the following result.

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 1      | Nail      | 10.00 | 2013-03-31 |
| 2      | Washer    | 15.00 | 2013-03-29 |
| 3      | Nut       | 15.00 | 2013-03-29 |
| 4      | Screw     | 25.00 | 2013-03-30 |
| 5      | Super_Nut | 30.00 | 2013-03-30 |
| 6      | New Nut   |  NULL | NULL       |
+--------+-----------+-------+------------+
```

Selecting Specific Columns

To query specific columns, list the columns in the SELECT clause. You write the columns in the order you want to see them in the output table. For example, the SELECT statement in Listing 2.2 queries the **p_name** and the **price** columns from the **product** table.

Listing 2.2: Querying specific columns

```
SELECT p_name, price FROM product;
```

All rows containing **p_name** and **price** columns will be returned:

```
+-----------+-------+
| p_name    | price |
+-----------+-------+
| Nail      | 10.00 |
| Washer    | 15.00 |
| Nut       | 15.00 |
| Screw     | 25.00 |
| Super_Nut | 30.00 |
| New Nut   |  NULL |
+-----------+-------+
```

Selecting Rows with WHERE

To query specific rows, use the WHERE clause. Recall that the SQL SELECT statement has the following syntax.

```
SELECT column_names FROM table [WHERE condition];
```

For example, the SQL statement in Listing 2.3 queries the **p_name** and **price** data from the **product** table with price = 15.

Listing 2.3: Querying specific rows

```
SELECT p_name, price FROM product WHERE price = 15;
```

Only rows whose price is 15 will be returned by the query, in this case the Washer and Nut. The query output is as follows.

```
+--------+-------+
| p_name | price |
+--------+-------+
| Washer | 15.00 |
| Nut    | 15.00 |
+--------+-------+
```

The equal sign (=) in the WHERE condition in Listing 2.3 is one of the comparison operators. Table 2.1 summarizes all comparison operators.

Operator	Description
=	Equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
!=	Not equal to

Table 2.1: Comparison operators

As another example, Listing 2.4 shows a WHERE clause that uses the not equal to (!=) operator.

Listing 2.4: Using the != comparison operator

```
SELECT p_name, price FROM product WHERE p_name != 'Nut';
```

Only rows whose **p_name** is not ‘Nut’ will be returned by the query. In this case, the query output will be as follows.

```
+-----------+-------+
| p_name    | price |
+-----------+-------+
| Nail      | 10.00 |
| Washer    | 15.00 |
| Screw     | 25.00 |
| Super_Nut | 30.00 |
| New Nut   |  NULL |
+-----------+-------+
```

Compound Conditions

The condition **p_name != 'Nut'** in Listing 2.4 is called a predicate. Using the AND and OR logical operator you can combine predicates to form a compound condition. Only rows that satisfy the compound condition will be returned by the query.

The rules for the OR logical operator are given in Table 2.2.

Left condition	Logical operator	Right condition	Compound condition
True	OR	True	True
True	OR	False	True
False	OR	True	True
False	OR	False	False

Table 2.2: The OR rules

In principle, the result of the OR compound condition is true (satisfying the condition) if any one of the two conditions being OR-ed is true; otherwise, if none of the conditions is true, the compound condition is false (not satisfying the condition).

The rules for the AND logical operator are presented in Table 2.3.

Left condition	Logical operator	Right condition	Compound condition
True	AND	True	True
True	AND	False	False
False	AND	True	False
False	AND	False	False

Table 2.3: The AND rules

Basically, the result of the AND compound condition is true only if the two conditions being AND-ed are true; otherwise, the result is false.

For example, the statement in Listing 2.5 contains three predicates in its WHERE clause.

Listing 2.5: A query with three predicates

```
SELECT *
FROM product
WHERE (launch_dt >= '2013-03-30'
OR price          > 15)
AND (p_name      != 'Nail');
```

The result of the first compound condition (launch_dt >= '30-MAR-13' OR price > 15) is true for Nail, Screw and Super_Nut rows in the product table; AND-ing this result with the (p_name != 'Nail') predicate results in two products, the Screw and Super_Nut.

Here is the output of the query in Listing 2.5:

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 4      | Screw     | 25.00 | 2013-03-30 |
| 5      | Super_Nut | 30.00 | 2013-03-30 |
+--------+-----------+-------+------------+
```

Note that New Nut does not satisfy the condition because applying any of the comparison operators to NULL results evaluates to false (the price and launch_dt of the New Nut are NULL). The section “Handling NULL” later in this chapter explains more about NULL.

Evaluation Precedence and the Use of Parentheses

If a compound condition contains both the OR condition and the AND condition, the AND condition will be evaluated first because AND has a higher precedence than OR. However, anything in parentheses will have an even higher precedence than AND. For example, the SELECT statement in Listing 2.5 has an OR and an AND, but the OR condition is in parentheses so the OR condition is evaluated first. If you remove the parentheses in the SELECT statement in Listing 2.5, the query will return a different result. Consider the statement in Listing 2.6, which is similar to that in Listing 2.5 except that the parentheses have been removed.

Listing 2.6: Evaluation precedence

```
SELECT *
FROM product
WHERE launch_dt >= '2013-03-30'
OR price          > 15
AND p_name      != 'Nail';
```

For your reading convenience, the **product** table is reprinted here.

```
P_CODE P_NAME      PRICE LAUNCH_DT
------ ---------- ------ ---------
1      Nail        10.00 31-MAR-13
2      Washer      15.00 29-MAR-13
3      Nut         15.00 29-MAR-13
4      Screw       25.00 30-MAR-13
5      Super_Nut   30.00 30-MAR-13
6      New Nut    NULL   NULL
```

Without the parentheses, the compound condition price > 15 AND p_name != 'Nail' will be evaluated first, resulting in the Screw and Super_Nut. The result is then OR-ed with the launch_dt >= 30-MAR-13' condition, resulting in these three rows.

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 1      | Nail      | 10.00 | 2013-03-31 |
| 4      | Screw     | 25.00 | 2013-03-30 |
| 5      | Super_Nut | 30.00 | 2013-03-30 |
+--------+-----------+-------+------------+
```

The NOT logical operator

You can use NOT to negate a condition and return rows that do not satisfy the condition. Consider the query in Listing 2.7.

Listing 2.7: Using the NOT operator

```
SELECT *
FROM product
WHERE NOT (launch_dt >= '2013-03-30'
OR price              > 15
AND p_name           != 'Nail' );
```

Thanks to the NOT operator in the query in Listing 2.7, the two rows not satisfying the condition in Listing 2.6 will now be returned.

```
+--------+--------+-------+------------+
| p_code | p_name | price | launch_dt  |
+--------+--------+-------+------------+
| 2      | Washer | 15.00 | 2013-03-29 |
| 3      | Nut    | 15.00 | 2013-03-29 |
+--------+--------+-------+------------+
```

As another example, the query in Listing 2.8 negates the last predicate only (as opposed to the previous query that negated the overall WHERE condition).

Listing 2.8: Using NOT on one predicate

```
SELECT *
FROM product
WHERE (launch_dt >= '2013-03-30'
OR price          > 15)
AND NOT (p_name  != 'Nail');
```

The output of the query in Listing 2.8 is as follows.

```
+--------+--------+-------+------------+
| p_code | p_name | price | launch_dt  |
+--------+--------+-------+------------+
| 1      | Nail   | 10.00 | 2013-03-31 |
+--------+--------+-------+------------+
```

The BETWEEN Operator

The BETWEEN operator evaluates equality to any value within a range. The range is specified by a boundary, which specifies the lowest and the highest values.

Here is the syntax for BETWEEN.

```
SELECT columns FROM table
WHERE column BETWEEN(lowest_value, highest_value);
```

The boundary values are inclusive, meaning *lowest_value* and *highest_value* will be included in the equality evaluation.

For example, the query in Listing 2.9 uses the BETWEEN operator to specify the lowest and highest prices that need to be returned from the product table.

Listing 2.9: Using the BETWEEN operator

```
SELECT * FROM product WHERE price BETWEEN 15 AND 25;
```

Here is the output of the query in Listing 2.9.

```
+--------+--------+-------+------------+
| p_code | p_name | price | launch_dt  |
+--------+--------+-------+------------+
| 2      | Washer | 15.00 | 2013-03-29 |
| 3      | Nut    | 15.00 | 2013-03-29 |
| 4      | Screw  | 25.00 | 2013-03-30 |
+--------+--------+-------+------------+
```

The IN Operator

The IN operator compares a column with a list of values. The syntax for a query that uses IN is as follows.

```
SELECT columns FROM table
WHERE column IN(value1, value2, ...);
```

For example, the query in Listing 2.10 uses the IN operator to select all columns whose price is in the list (10, 25, 50).

Listing 2.10: Using the IN operator

```
SELECT * FROM product WHERE price IN (10, 25, 50);
```

The output of the query in Listing 2.10 is as follows.

```
+--------+--------+-------+------------+
| p_code | p_name | price | launch_dt  |
+--------+--------+-------+------------+
| 1      | Nail   | 10.00 | 2013-03-31 |
| 4      | Screw  | 25.00 | 2013-03-30 |
+--------+--------+-------+------------+
```

The LIKE Operator

The LIKE operator allows you to specify an imprecise equality condition. The syntax is as follows.

```
SELECT columns FROM table
WHERE column LIKE ' ... wildcard_character ... ';
```

The wildcard character can be a percentage sign (%) to represent any number of characters or an underscore (_) to represent a single occurrence of any character.

As an example, the query in Listing 2.11 uses the LIKE operator to find products whose name starts with N and is followed by two other characters plus products whose name starts with Sc and can be of any length.

Listing 2.11: Using the LIKE operator

```
SELECT * FROM product WHERE p_name LIKE 'N__' OR p_name LIKE 'Sc%';
```

The output of the query in Listing 2.11 is this.

```
+--------+--------+-------+------------+
| p_code | p_name | price | launch_dt  |
+--------+--------+-------+------------+
| 3      | Nut    | 15.00 | 2013-03-29 |
| 4      | Screw  | 25.00 | 2013-03-30 |
+--------+--------+-------+------------+
```

Even though you can use LIKE for numeric columns, it is primarily used with columns of type string.

Escaping the Wildcard Character

If the string you specify in the LIKE operator contains an underscore or a percentage sign, SQL will regard it as a wild character. For example, if you want to query products that have an underscore in their names, your SQL statement would look like that in Listing 2.12.

Listing 2.12: A wildcard character _ in the LIKE string

```
SELECT * FROM product WHERE p_name LIKE '%_%';
```

If you execute the query in Listing 2.12, the query will return all rows instead of just the Super_Nut, because the underscore in the LIKE operator is regarded as a wild card character, i.e. any one character. Listing 2.13 resolves this problem by prefixing the wild card character with the \ (backslash) escape character, meaning any character in the LIKE operator after a backslash will be considered a character, not as a wildcard character. Now only rows whose **p_name** contains an underscore will be returned.

Listing 2.13: Escaping the wildcard character _

```
SELECT * FROM product WHERE p_name LIKE '%\_%';
```

The query in Listing 2.13 will produce the following output.

```
+--------+---------+-------+-----------+
| p_code | p_name  | price | launch_dt |
+--------+---------+-------+-----------+
| 6      | New Nut |  NULL | NULL      |
+--------+---------+-------+-----------+
```

Combining the NOT operator

You can combine NOT with BETWEEN, IN, or LIKE to negate their conditions. For example, the query in Listing 2.14 uses NOT with BETWEEN.

Listing 2.14: Using NOT with BETWEEN

```
SELECT * FROM product WHERE price NOT BETWEEN 15 AND 25;
```

Executing the query in Listing 2.14 will give you this result.

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 1      | Nail      | 10.00 | 2013-03-31 |
| 5      | Super_Nut | 30.00 | 2013-03-30 |
+--------+-----------+-------+------------+
```

Handling NULLNULL, an SQL reserved word, represents the absence of data. NULL is applicable to any data type. It is not the same as a numeric zero or an empty string or a 0000/00/00 date. You can specify whether or not a column can be null in the CREATE TABLE statement for creating the table.

The result of applying any of the comparison operators on NULL is always NULL. You can only test whether or not a column is NULL by using the IS NULL or IS NOT NULL operator.

Consider the query in Listing 2.15.

Listing 2.15: Invalid usage of the equal operator on NULL

```
SELECT * FROM product WHERE price = NULL;
```

Executing the query in Listing 2.15 produces no output. In fact, you will get the following message.

```
Empty set (0.00 sec)
```

As another example, consider the query in Listing 2.16 that uses IS NULL.

Listing 2.16: Using IS NULL

```
SELECT * FROM product WHERE price IS NULL;
```

The query output is as follows.

```
+--------+---------+-------+-----------+
| p_code | p_name  | price | launch_dt |
+--------+---------+-------+-----------+
| 6      | New Nut |  NULL | NULL      |
+--------+---------+-------+-----------+
```

Note

Chapter 10, “Built-in Functions,” discusses functions that you can use to test column nullity.

Summary

In this chapter you learned the basics queries using the SELECT statement. In the next chapter you will learn an advanced query feature called full-text search.

Chapter 3
Full-Text Search

A full-text search is a query for finding a word or string in textual columns (columns with the string data type). In MySQL you use the MATCH function to perform a full-text search.

There are three modes of searches you can perform using the MATCH function: Natural Language, Query Expansion, and Boolean. This chapter shows you how to do full-text searches using the MATCH function.

Natural Language Searches

In Chapter 2, "Basic Queries" you learned that you could query a string type column by specifying a search string using the LIKE operator in the WHERE condition. For example, the query in Listing 3.1 searches the **product** table for rows that contain the search string "nut" in the **p_descr** column.

Listing 3.1: Searching for "nut" using the LIKE operator

```
SELECT * FROM product WHERE p_descr LIKE '%nut%';
```

Executing the query against the following **product** table

```
+--------+-----------+------------------------------------------+
| p_code | p_name    | p_descr                                  |
+--------+-----------+------------------------------------------+
| 1      | Nail      | A nail is a pin-shaped object of metal   |
| 2      | Washer    | A washer is a thin plate with a hole     |
| 3      | Nut       | A nut is a fastener with a threaded hole |
| 4      | Screw     | A screw, or bolt, is a type of fastener  |
| 5      | Super_Nut | The most expensive nut in the market     |
| 6      | New Nut   | The newer nut we introduced in May 2013  |
+--------+-----------+------------------------------------------+
```

returns the following output rows

```
+--------+-----------+------------------------------------------+
| p_code | p_name    | p_descr                                  |
+--------+-----------+------------------------------------------+
| 3      | Nut       | A nut is a fastener with a threaded hole |
| 5      | Super_Nut | The most expensive nut in the market     |
| 6      | New Nut   | The newer nut we introduced in May 2013  |
+--------+-----------+------------------------------------------+
```

If the **p_descr** column contains very long strings and the table has a lot of rows, using the LIKE operator will likely be slow. The faster solution is to do a full-text search.

The magic behind fast full-text searches in MySQL is its full-text indexing. A full-text index compiles every string (word) stored in the indexed column(s). For example, the strings “nail”, “pin”, “shaped”, “object”, “metal”, “washer”, and so on, in the **p_descr** column of the **product** table, are indexed.

Before you can do a full-text search on a column, you must first create a full-text index on that column. The syntax for creating a full-text index is as follows.

```
CREATE FULLTEXT INDEX index_name ON table(columns);
```

For example, Listing 3.2 creates a full-text index on the **p_descr** column of the **product** table.

Listing 3.2: Creating a full-text index

```
CREATE FULLTEXT INDEX p_descr_idx ON product(p_descr);
```

Once you have a full-text index, you can use the MATCH function to query it. The syntax for a full-text search in the natural language mode is as follows.

```
SELECT columns
FROM table
WHERE MATCH (search_column) AGAINST (search_string IN NATURAL
      LANGUAGE MODE);
```

As an example, the query in Listing 3.3 uses the MATCH function to search for “nut.”

Listing 3.3: Searching for “nut” with the MATCH function

```
SELECT *
FROM product
WHERE MATCH (p_descr) AGAINST ('nut' IN NATURAL LANGUAGE MODE);
```

The output of this query, which is the same as the output of the query in Listing 3.1, is as follows.

```
+--------+-----------+-------------------------------------------+
| p_code | p_name    | p_descr                                   |
+--------+-----------+-------------------------------------------+
| 3      | Nut       | A nut is a fastener with a threaded hole |
| 5      | Super_Nut | The most expensive nut in the market     |
| 6      | New Nut   | The newer nut we introduced in May 2013  |
+--------+-----------+-------------------------------------------+
```

The natural language search ignores stopwords like “a”, “and”, “the”, “in”, and “with”. For example, the query in Listing 3.4, where the search string is “in”, a stopword, will return zero row.

Listing 3.4: Searching for “in”

```
SELECT *
FROM product
WHERE MATCH (p_descr) AGAINST ('in' IN NATURAL LANGUAGE MODE);
```

The result will be an empty set because “in” is a stopword.

You can write a query with a compound condition mixing a MATCH with another MATCH, or a MATCH with any of the condition expression you learned in the previous chapters. Listing 3.5, for example, mixes MATCH with LIKE.

Listing 3.5: Mixing MATCH with LIKE

```
SELECT *
FROM product
WHERE MATCH (p_descr) AGAINST ('in' IN NATURAL LANGUAGE MODE)
OR p_descr LIKE '%fastener%';
```

Here is the output of this query.

```
+--------+--------+------------------------------------------+
| p_code | p_name | p_descr                                  |
+--------+--------+------------------------------------------+
| 3      | Nut    | A nut is a fastener with a threaded hole |
| 4      | Screw  | A screw, or bolt, is a type of fastener  |
+--------+--------+------------------------------------------+
Query Expansion Searches
```

The MATCH function can be extended with a WITH QUERY EXPANSION modifier. The syntax will then look like the following.

```
SELECT columns
FROM table
WHERE MATCH (search_column) AGAINST (search_string WITH QUERY
       EXPANSION);
```

With query expansion the MATCH function works in two steps:

- Rows are returned whenever their searched columns contain the search string
- Each word in the searched columns in the retrieved rows is then used as a search string. These derived search strings are called expanded search strings and are used to search the specified columns to find more rows.

For example, the query in Listing 3.6 modifies the query in Listing 3.2, adding the query expansion modifier.

Listing 3.6: Query Expansion modifier

```
SELECT *
FROM product
WHERE MATCH (p_descr) AGAINST ('nut' WITH QUERY EXPANSION);
```

The output rows will now be as follows.

```
+--------+-----------+-----------------------------------------+
| p_code | p_name    | p_descr                                 |
+--------+-----------+-----------------------------------------+
| 6      | New Nut   | The newer nut we introduced in May 2013  |
| 5      | Super_Nut | The most expensive nut in the market     |
| 3      | Nut       | A nut is a fastener with a threaded hole |
| 2      | Washer    | A washer is a thin plate with a hole     |
| 4      | Screw     | A screw, or bolt, is a type of fastener  |
+--------+-----------+-----------------------------------------+
```

The query in Listing 3.2, which did not use query expansion, returned only rows with p_code = 3, 5 and 6. The query in Listing 3.6 also returns the row with p_code = 2 because the word “hole” in the row with p_code = 3 was used as one of the expanded search strings. By the same token, the row with p_code = 4 is returned thanks to the word “fastener” in the row with p_code = 3.

Boolean Searches

In Boolean full-text searches, certain characters have special meaning at the beginning or end of words in the search string. The syntax for the Boolean search is as follows.

```
SELECT columns
FROM table
WHERE MATCH (search_column) AGAINST (search_string IN BOOLEAN MODE);
```

In the query in Listing 3.7, for example, the +and − operators indicate that the word “nut” must be present and the word “fastener” must be absent. Thus, the query will return rows that contain the word “nut” but not “fastener”.

Listing 3.7: A Boolean search

```
SELECT *
FROM product
WHERE MATCH (p_descr) AGAINST ('+nut -fastener' IN BOOLEAN MODE);
```

Here is the output of the query. Note that the row with p_code = 3 is not returned, because it contains both “nut” and “fastener”.

```
+--------+-----------+------------------------------------------+
| p_code | p_name    | p_descr                                  |
+--------+-----------+------------------------------------------+
| 5      | Super_Nut | The most expensive nut in the market     |
| 6      | New Nut   | The newer nut we introduced in May 2013 |
+--------+-----------+------------------------------------------+
```

A pair of double quotes can be used in a Boolean search to indicate that the words they enclose should be considered a single search string. The query in Listing 3.8 is an example query that employs double quotes in a Boolean search. The query returns the row with p_code = 3 only as it is the only row that contains the phrase "threaded hole" in its **p_descr** column. The row with p_code = 2 is not returned as it only contains the word "hole".

Listing 3.8: Using double quotes in a Boolean search

```
SELECT *
FROM product
WHERE MATCH (p_descr) AGAINST ('"threaded hole"' IN BOOLEAN MODE);
```

Here is the query output.

```
+--------+--------+------------------------------------------+
| p_code | p_name | p_descr                                  |
+--------+--------+------------------------------------------+
| 3      | Nut    | A nut is a fastener with a threaded hole |
+--------+--------+------------------------------------------+
```

The asterisk (*) is another Boolean character. It indicates that only the characters preceding it should be used in the search. The query in Listing 3.9 uses the asterisk. It will find all words that starts with "ma" in the **p_descr** column.

Listing 3.9: Boolean wildcard

```
SELECT *
FROM product
WHERE MATCH (p_descr) AGAINST ('ma*' IN BOOLEAN MODE);
```

The query returns the following output.

```
+--------+-----------+------------------------------------------+
| p_code | p_name    | p_descr                                  |
+--------+-----------+------------------------------------------+
| 5      | Super_Nut | The most expensive nut in the market     |
| 6      | New Nut   | The newer nut we introduced in May 2013 |
+--------+-----------+------------------------------------------+
```

As the search is not case sensitive, “market” and “May” are considered matches.

Summary

In this chapter you learned how to use the MATCH function to do a full-text search. In the next chapter you will learn how to format query outputs.

Chapter 4 Query Output

All the queries in Chapter 2, “Basic Queries” returned rows that contained columns from the source table. However, output rows can also contain string or numeric expressions that include string or numeric literals, operators, and functions.

In this chapter you learn how to manipulate query output using expressions and how to order and store output rows into a table.

Column Aliases

By default the names of the output columns in the query output are the names of the columns of the queried table. However, you don’t have to be stuck with the original column names. You can give them different names or aliases if you wish.

The syntax for the SELECT clause that uses aliases is as follows.

```
SELECT column_1 AS alias1, column_2 AS alias2, ...
FROM table;
```

An alias can consist of one or multiple words. You must enclose a multiword alias with quotes, e.g. “PRODUCT NAME”. For example, the query in Listing 4.1 uses an alias for the **p_name** column.

Listing 4.1: Using an alias in a query

```
SELECT p_code,
  p_name AS "PRODUCT NAME"
FROM product;
```

When you execute the query in Listing 4.1, you will see the output rows like this.

```
+--------+--------------+
| p_code | PRODUCT NAME |
+--------+--------------+
| 1      | Nail         |
| 2      | Washer       |
| 3      | Nut          |
| 4      | Screw        |
| 5      | Super_Nut    |
| 6      | New Nut      |
+--------+--------------+
```

Expressions

An output column can also be an expression. An expression in the SELECT clause can include columns, literal values, arithmetic or string operators, and functions. For instance, the SELECT clause in the query in Listing 4.2 employs several expressions.

Listing 4.2: Various types of output columns

```
SELECT p_code,
  CONCAT('In Uppercase: '
  , UPPER(p_name))                 AS "PRODUCT NAME",
  (price * 100)                    AS "PRICE*100",
  DATE_FORMAT(launch_dt, '%d/%m/%Y') AS "LAUNCH_DATE"
FROM product;
```

The output of the query in Listing 4.2 will have four columns.

The first output column, **p_code**, is a column from the product table.

The second output column (aliased "PRODUCT NAME") is an expression that contains three parts, a literal 'p_name in Uppercase: ', a concatenation string operator (||), and UPPER(p_name). The latter, UPPER, is a function applied to the p_name column from the product table. The UPPER function changes the case of the product names to uppercase.

The third output column ("NORMALIZED_PRICE") is an arithmetic expression (price*100).

The last output column ("LAUNCH_DATE") is the **launch_date** column formatted as DD/MM/YYYY.

Applied against the following product table

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 1      | Nail      | 10.00 | 2013-03-31 |
| 2      | Washer    | 15.00 | 2013-03-29 |
| 3      | Nut       | 15.00 | 2013-03-29 |
| 4      | Screw     | 25.00 | 2013-03-30 |
| 5      | Super_Nut | 30.00 | 2013-03-30 |
| 6      | New Nut   |  NULL | NULL       |
+--------+-----------+-------+------------+
```

the query in Listing 4.2 returns the following rows.

```
+--------+-------------------------+-----------+-------------+
| p_code | PRODUCT NAME            | PRICE*100 | LAUNCH_DATE |
+--------+-------------------------+-----------+-------------+
| 1      | In Uppercase: NAIL      |   1000.00 | 31/03/2013  |
| 2      | In Uppercase: WASHER    |   1500.00 | 29/03/2013  |
| 3      | In Uppercase: NUT       |   1500.00 | 29/03/2013  |
| 4      | In Uppercase: SCREW     |   2500.00 | 30/03/2013  |
| 5      | In Uppercase: SUPER_NUT |   3000.00 | 30/03/2013  |
| 6      | In Uppercase: NEW NUT   |      NULL | NULL        |
+--------+-------------------------+-----------+-------------+
```

You can use other arithmetic operators in addition to the multiplication (*) operator in your column. These include addition (+), subtraction (-), and division (/)

Note
Chapter 10, "Built-in Functions" explains functions in more detail.

Limiting the Number of Rows

You can limit the number of output row by using the LIMIT clause. Its syntax is as follows.

```
SELECT columns FROM table(s)
WHERE conditions LIMIT count;
```

Here *count* is the maximum number of output rows returned by the query.

As an example, take a look at the query in Listing 4.3.

Listing 4.3: Using LIMIT

```
SELECT * FROM product WHERE price > 10 LIMIT 3;
```

Without the clause **LIMIT 3**, the number of output rows would be 4. On the other hand, the query in Listing 4.3 returns these three rows.

```
+--------+--------+-------+------------+
| p_code | p_name | price | launch_dt  |
+--------+--------+-------+------------+
| 2      | Washer | 15.00 | 2013-03-29 |
| 3      | Nut    | 15.00 | 2013-03-29 |
| 4      | Screw  | 25.00 | 2013-03-30 |
+--------+--------+-------+------------+
```

The DISTINCT Keyword

A query may return duplicate rows. Two rows are duplicates if each of their columns contains exactly the same data. If you don’t want to see duplicate output rows, use DISTINCT in your SELECT clause. You can use DISTINCT on one column or multiple columns.

Using DISTINCT on A Single Column

The query in Listing 4.4 uses DISTINCT on the price column.

Listing 4.4: Using DISTINCT on a single column

```
SELECT DISTINCT price FROM product ORDER BY price;
```

Without DISTINCT, the query in Listing 4.4 will return six rows that include two duplicate prices for row 2 and row 3. Instead, the query in Listing 4.4 returns the following output.

```
+-------+
| price |
+-------+
|  NULL |
| 10.00 |
| 15.00 |
| 25.00 |
| 30.00 |
+-------+
```

Using DISTINCT on Multiple Columns

If a query returns multiple columns, two rows are considered duplicates if all their columns have the same values. They are not duplicates if only one column has the same value.

The DISTINCT keyword can be applied on multiple columns too. For example, the query in Listing 4.5 uses DISTINCT on multiple columns.

Listing 4.5: Using DISTINCT on multiple columns

```
SELECT DISTINCT price, launch_dt FROM product ORDER BY price;
```

Here is the output. Note that output rows with the same **price** and **launch_dt** will only be shown once.

```
+-------+------------+
| price | launch_dt  |
+-------+------------+
|  NULL | NULL       |
| 10.00 | 2013-03-31 |
| 15.00 | 2013-03-29 |
| 25.00 | 2013-03-30 |
| 30.00 | 2013-03-30 |
+-------+------------+
```

Aggregate Functions

You can manipulate your query output further by using aggregate functions. The aggregate functions are summarized in Table 4.1.

Function	Description
MAX(column)	The maximum column value
MIN(column)	The minimum column value
SUM(column)	The sum of column values
AVG(column)	The average column value
COUNT(column)	The count of rows
COUNT(*)	The count of all rows including NULL.

Table 4.1: Built-in aggregate functions

As an example, the query in Listing 4.6 uses the aggregate functions in Table 4.1.

Listing 4.6: Using aggregate functions

```
SELECT MAX(price) 'max',
  MIN(price) 'min',
  SUM(price) 'sum',
  AVG(price) 'avg',
  COUNT(price) 'count(price)',
  COUNT(*) 'count(*)'
FROM product;
```

Note that only COUNT(*) takes into account the New Nut product because its price is NULL.

The output of the query in Listing 4.6 is this.

```
+-------+-------+-------+-----------+--------------+----------+
| max   | min   | sum   | avg       | count(price) | count(*) |
+-------+-------+-------+-----------+--------------+----------+
| 30.00 | 10.00 | 95.00 | 19.000000 |            5 |        6 |
+-------+-------+-------+-----------+--------------+----------+
```

The CASE Expression

CASE allows you to have dynamic query output in which a column value may vary depending on the value of the column. CASE comes in two flavors: Simple and Searched. Both will be explained in the following subsections.

The Simple CASE

The general syntax for the Simple CASE is as follows.

```
SELECT columns,
  CASE column
    WHEN equal_value1
    THEN output_value1
    WHEN equal_value2
    THEN output_value2
    WHEN ...
    [ELSE else_value]
  END AS output_column
FROM table
WHERE ... ;
```

In the Simple CASE, *column_name* is compared to *equal_value*s in the WHEN clause, starting from the first WHEN down to the last WHEN. If *column_name* matches a WHEN value, the value right after the THEN clause is returned and the CASE process stops. If *column_name* matches none of the WHEN values, *else_value* is returned if there exists an ELSE clause. If *column_name* matches none of the WHEN values but no ELSE clause exists, NULL will be returned.

As an example, the query in Listing 4.7 uses a Simple CASE expression for the **price** column to produce a **price_cat** (price category) output column.

Listing 4.7: An example of the Simple CASE

```
SELECT p_code,
  p_name,
  CASE price
    WHEN 10
    THEN 'Cheap'
    WHEN 15
    THEN 'Medium'
    WHEN 25
    THEN 'Expensive'
    ELSE 'Others'
  END AS price_cat
FROM product;
```

Assuming the product table has the following data

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 1      | Nail      | 10.00 | 2013-03-31 |
| 2      | Washer    | 15.00 | 2013-03-29 |
| 3      | Nut       | 15.00 | 2013-03-29 |
| 4      | Screw     | 25.00 | 2013-03-30 |
| 5      | Super_Nut | 30.00 | 2013-03-30 |
| 6      | New Nut   |  NULL | NULL       |
+--------+-----------+-------+------------+
```

the query will return these rows.

```
+--------+-----------+-----------+
| p_code | p_name    | price_cat |
+--------+-----------+-----------+
| 1      | Nail      | Cheap     |
| 2      | Washer    | Medium    |
| 3      | Nut       | Medium    |
| 4      | Screw     | Expensive |
| 5      | Super_Nut | Others    |
| 6      | New Nut   | Others    |
+--------+-----------+-----------+
```

The Searched CASE

The case in the Simple CASE compares a column with various values. On the hand, the case in the Searched CASE can be any condition. Here is the syntax for the Searched CASE.

```
SELECT columns,
  CASE
    WHEN condition1
    THEN output_value1
    WHEN condition2
    THEN output_value2
    WHEN ...
    ELSE else_value
  END AS output_column
FROM table
WHERE ... ;
```

The conditions are evaluated starting from the first WHEN down to the last WHEN. If a WHEN condition is met, its THEN output_value is returned to the output_column and the CASE process stops. If none of the WHEN conditions is met, *else_value* is returned if there exists an ELSE clause. If no condition is met and no ELSE clause exists, NULL will be returned.

For instance, the query in Listing 4.8 uses a Searched CASE. While the Simple CASE in Listing 4.7 categorized the products based on only their prices, this Searched CASE categorizes the products based on the various conditions which can involve more than just the price. Note that in the Search CASE, NULL equality can be a condition, something that is not allowed in the Simple CASE.

Listing 4.8: An example of the Searched CASE

```
SELECT p_code,
  p_name,
  CASE
    WHEN (price <= 10
    AND p_name NOT LIKE 'Nut%')
    THEN 'Cheap'
    WHEN price BETWEEN 11 AND 25
    THEN 'Medium'
    WHEN price > 25 and launch_dt> '2013-03-29'
    THEN 'Expensive'
    WHEN price IS NULL
    THEN 'Not valid'
    ELSE 'Others'
  END AS product_cat
, launch_dt
FROM product;
```

Applying the query against the following product table

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 1      | Nail      | 10.00 | 2013-03-31 |
| 2      | Washer    | 15.00 | 2013-03-29 |
| 3      | Nut       | 15.00 | 2013-03-29 |
| 4      | Screw     | 25.00 | 2013-03-30 |
| 5      | Super_Nut | 30.00 | 2013-03-30 |
| 6      | New Nut   |  NULL | NULL       |
+--------+-----------+-------+------------+
```

will return the following rows.

```
+--------+----------+------------+-----------+
| p_code | p_name   | product_cat | launch_dt  |
+--------+----------+------------+-----------+
| 1      | Nail     | Cheap       | 2013-03-31 |
| 2      | Washer   | Medium      | 2013-03-29 |
| 3      | Nut      | Medium      | 2013-03-29 |
| 4      | Screw    | Medium      | 2013-03-30 |
| 5      | Super_Nut | Expensive   | 2013-03-30 |
| 6      | New Nut  | Not valid   | NULL       |
+--------+----------+------------+-----------+
```

Ordering Output Rows

To provide better visualization of the output, you can order output rows based on certain criteria. To order the output, use the ORDER BY clause. The ORDER BY clause must appear last in a SELECT statement.

Here is the syntax for a query having the ORDER BY clause.

```
SELECT columns FROM
table
WHERE condition ORDER BY column(s)
```

You can order output rows in one of the following methods.

- by one or more columns
- in ascending or descending direction
- by using the GROUP BY clause
- by using UNION and other set operators

Each of the methods is explained in the subsections below.

Ordering by One Column

To order your query output rows, use the ORDER BY clause with one column. For instance, have a look at the query in Listing 4.9.

Listing 4.9: Ordering by one column

```
SELECT * FROM product ORDER BY p_name;
```

When you apply the query against the following product table

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 1      | Nail      | 10.00 | 2013-03-31 |
| 2      | Washer    | 15.00 | 2013-03-29 |
| 3      | Nut       | 15.00 | 2013-03-29 |
| 4      | Screw     | 25.00 | 2013-03-30 |
| 5      | Super_Nut | 30.00 | 2013-03-30 |
| 6      | New Nut   |  NULL | NULL       |
+--------+-----------+-------+------------+
```

you will see the following output.

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 1      | Nail      | 10.00 | 2013-03-31 |
| 6      | New Nut   |  NULL | NULL       |
| 3      | Nut       | 15.00 | 2013-03-29 |
| 4      | Screw     | 25.00 | 2013-03-30 |
| 5      | Super_Nut | 30.00 | 2013-03-30 |
| 2      | Washer    | 15.00 | 2013-03-29 |
+--------+-----------+-------+------------+
```

Direction of Order

The default direction is ascending. To order a column in descending direction, use the DESC reserved word. For example, the query in Listing 4.10 is similar to that in Listing 4.9 except that the output is presented in descending order.

Listing 4.10: Changing the order direction

```
SELECT * FROM product ORDER BY p_name DESC;
```

The output rows will be returned with p_name sorted in descending order.

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 2      | Washer    | 15.00 | 2013-03-29 |
| 5      | Super_Nut | 30.00 | 2013-03-30 |
| 4      | Screw     | 25.00 | 2013-03-30 |
| 3      | Nut       | 15.00 | 2013-03-29 |
| 6      | New Nut   |  NULL | NULL       |
| 1      | Nail      | 10.00 | 2013-03-31 |
+--------+-----------+-------+------------+
```

Multiple Columns

To order by more than one column, list the columns in the ORDER BY clause. The sequence of columns listed is significant. The order will be conducted by the first column in the list, followed by the second column, and so on. For example, if the ORDER BY clause has two columns, the query output will first be ordered by the first column. Any rows with identical values in the first column will be further ordered by the second column.

For example, the query in Listing 4.11 uses an ORDER BY clause with two columns.

Listing 4.11: Multiple column ordering

```
SELECT * FROM product ORDER BY launch_dt, price;
```

Applying the query against the product table.

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 1      | Nail      | 10.00 | 2013-03-31 |
| 2      | Washer    | 15.00 | 2013-03-29 |
| 3      | Nut       | 15.00 | 2013-03-29 |
| 4      | Screw     | 25.00 | 2013-03-30 |
| 5      | Super_Nut | 30.00 | 2013-03-30 |
| 6      | New Nut   |  NULL | NULL       |
+--------+-----------+-------+------------+
```

The output rows will first be ordered by launch_dt and then by price, both in ascending order. The secondary ordering by price is seen on the Screw and Super_Nut rows. Their launch_dt’s are the same, 30-MAR-13. Their prices are different, Screw's lower than Super_Nut's, hence Screw row comes before the Super_Nut.

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 6      | New Nut   |  NULL | NULL       |
| 2      | Washer    | 15.00 | 2013-03-29 |
| 3      | Nut       | 15.00 | 2013-03-29 |
| 4      | Screw     | 25.00 | 2013-03-30 |
| 5      | Super_Nut | 30.00 | 2013-03-30 |
| 1      | Nail      | 10.00 | 2013-03-31 |
+--------+-----------+-------+------------+
```

Different Directions on Different Columns

You can apply different order directions on ordered columns too. For example, the query in Listing 4.12 uses different directions on different columns in its ORDER BY clause.

Listing 4.12: Using multiple directions of ORDER

```
SELECT * FROM product ORDER BY launch_dt, price DESC;
```

Applying the query against the product table, the output rows will be ordered by launch_dt in ascending order and then by price in descending order. Now, the Super_Nut comes before the Screw.

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 6      | New Nut   |  NULL | NULL       |
| 2      | Washer    | 15.00 | 2013-03-29 |
| 3      | Nut       | 15.00 | 2013-03-29 |
| 5      | Super_Nut | 30.00 | 2013-03-30 |
| 4      | Screw     | 25.00 | 2013-03-30 |
| 1      | Nail      | 10.00 | 2013-03-31 |
+--------+-----------+-------+------------+
```

Ordering with a WHERE clause

If your SELECT statement has both the WHERE clause and the ORDER BY clause, ORDER BY must appear after the WHERE clause.

For example, the query in Listing 4.13 has both WHERE and ORDER BY. This query will return only Nut products.

Listing 4.13: Using both WHERE and ORDER BY

```
SELECT * FROM product WHERE p_name = 'Nut'
ORDER BY p_name, p_code DESC;
```

If you execute the query, you will see one row only, the Nut, in the output window.

```
+--------+--------+-------+------------+
| p_code | p_name | price | launch_dt  |
+--------+--------+-------+------------+
| 3      | Nut    | 15.00 | 2013-03-29 |
+--------+--------+-------+------------+
```

Storing Query Output

You can store the query output into a new or an existing table. To store the query output in a new table, use the following statement:

```
CREATE TABLE new_table AS SELECT ... ;
```

For instance, the query in Listing 4.14 executes a SELECT statement and stores its result in a new table called **nut_product**.

Listing 4.14: Storing output into a new table

```
CREATE TABLE nut_product AS
SELECT * FROM product WHERE p_name LIKE '%Nut%';
```

Applied against the **product** table, the query in Listing 4.14 will create a **nut_product** table with the following content.

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 3      | Nut       | 15.00 | 2013-03-29 |
| 5      | Super_Nut | 30.00 | 2013-03-30 |
| 6      | New Nut   |  NULL | NULL       |
+--------+-----------+-------+------------+
```

To store a query output into an existing table, use this syntax.

```
INSERT INTO existing_table AS SELECT ... ;
```

For example, the query in Listing 4.15 stores the query result in an existing table.

Listing 4.15: Storing output into an existing table

```
INSERT INTO non_nut
SELECT * FROM product WHERE p_name NOT LIKE '%Nut%';
```

Before executing INSERT statement of Listing 4.15, first you have to create a non_nut table by executing the following statement.

```
CREATE TABLE non_nut
  (
    p_code    VARCHAR(6),
    p_name    VARCHAR(15),
    price     DECIMAL(4,2),
    launch_dt DATE,
    PRIMARY KEY (p_code)
  );
```

Applying the query in Listing 4.15 against this **product** table

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 1      | Nail      | 10.00 | 2013-03-31 |
| 2      | Washer    | 15.00 | 2013-03-29 |
| 3      | Nut       | 15.00 | 2013-03-29 |
| 4      | Screw     | 25.00 | 2013-03-30 |
| 5      | Super_Nut | 30.00 | 2013-03-30 |
| 6      | New Nut   |  NULL | NULL       |
+--------+-----------+-------+------------+
```

gives you a **non_nut** table with the following rows.

```
+--------+--------+-------+------------+
| p_code | p_name | price | launch_dt  |
+--------+--------+-------+------------+
| 1      | Nail   | 10.00 | 2013-03-31 |
| 2      | Washer | 15.00 | 2013-03-29 |
| 4      | Screw  | 25.00 | 2013-03-30 |
+--------+--------+-------+------------+
```

Summary

SQL allows you to retrieve rows from a table and manipulate the output. You learned in this chapter that you can create aliases, use aggregate functions, and order rows.

Chapter 5 Grouping

A group is a set of rows having the same value on specific columns. In Chapter 4, “Query Output” you learned how to apply aggregate functions on all output rows. In this chapter you will learn how to create groups and apply aggregate functions on those groups.

The GROUP BY Clause

In a query the GROUP BY clause appears after the WHERE clause and before the ORDER BY clause, if any. Here is the syntax for the SELECT statement with WHERE, GROUP BY, and ORDER BY clauses.

```
SELECT columns,
  aggregate_function(group_columns)
FROM table(s)
WHERE condition
GROUP BY group_columns
ORDER BY column(s);
```

As an example, the query in Listing 5.1 groups the output from the **product** table by their launch date.

Listing 5.1: Grouping on one column

```
SELECT launch_dt,
  MAX(price) MAX ,
  MIN(price) MIN ,
  SUM(price) SUM ,
  AVG(price) AVG ,
  COUNT(price) CNT ,
  COUNT(*) AS 'C(*)'
FROM product
GROUP BY launch_dt
```

```
ORDER BY launch_dt;
```

Applied against a **product** table with the following rows, aggregations will be done by the four grouped launch dates: 29, 30 and 31 of March 2013, and NULL.

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 1      | Nail      | 10.00 | 2013-03-31 |
| 2      | Washer    | 15.00 | 2013-03-29 |
| 3      | Nut       | 15.00 | 2013-03-29 |
| 4      | Screw     | 25.00 | 2013-03-30 |
| 5      | Super_Nut | 30.00 | 2013-03-30 |
| 6      | New Nut   |  NULL | NULL       |
+--------+-----------+-------+------------+
```

The query output will have four rows, one for each of the four grouped launch dates. Note that the COUNT(price) element, which counts the rows with a value on their price column, produces 0. On the other hand, the COUNT(*) element, which counts the NULL launch dates, produces 1.

```
+------------+-------+-------+-------+-----------+-----+------+
| launch_dt  | MAX   | MIN   | SUM   | AVG       | CNT | C(*) |
+------------+-------+-------+-------+-----------+-----+------+
| NULL       |  NULL |  NULL |  NULL |      NULL |   0 |    1 |
| 2013-03-29 | 15.00 | 15.00 | 30.00 | 15.000000 |   2 |    2 |
| 2013-03-30 | 30.00 | 25.00 | 55.00 | 27.500000 |   2 |    2 |
| 2013-03-31 | 10.00 | 10.00 | 10.00 | 10.000000 |   1 |    1 |
+------------+-------+-------+-------+-----------+-----+------+
```

You can group by more than one column. If you do that, rows having the same value on all the columns will form a group. As an example, the query in Listing 5.2 groups rows by price and launch date.

Listing 5.2: Grouping on two columns

```
SELECT launch_dt,
  MAX(price) MAX ,
  MIN(price) MIN ,
  SUM(price) SUM ,
  AVG(price) AVG ,
  COUNT(price) CNT ,
  COUNT(*) AS 'C(*)'
FROM product
GROUP BY price,
  launch_dt
ORDER BY price,
  launch_dt;
```

Applied to the same **product** table, the output will have five rows. Even though the Screw and Super_Nut have the same price, they have different launch dates, and therefore form different groups.

```
+------------+-------+-------+-------+-----------+-----+------+
| launch_dt  | MAX   | MIN   | SUM   | AVG       | CNT | C(*) |
+------------+-------+-------+-------+-----------+-----+------+
| NULL       |  NULL |  NULL |  NULL |      NULL |   0 |    1 |
| 2013-03-31 | 10.00 | 10.00 | 10.00 | 10.000000 |   1 |    1 |
| 2013-03-29 | 15.00 | 15.00 | 30.00 | 15.000000 |   2 |    2 |
| 2013-03-30 | 25.00 | 25.00 | 25.00 | 25.000000 |   1 |    1 |
| 2013-03-30 | 30.00 | 30.00 | 30.00 | 30.000000 |   1 |    1 |
+------------+-------+-------+-------+-----------+-----+------+
```

The HAVING Keyword

The WHERE condition can be used to select individual rows. On the other hand, the HAVING condition is used to select individual groups. Only groups that satisfy the condition in the HAVING clause will be returned by the query. In other words, the HAVING condition is on the aggregate, not on a column.

If present, the HAVING clause must appear after the GROUP BY clause, as in the following syntax.

```
SELECT columns,
  aggregate_function(group_columns)
FROM table(s)
WHERE condition
GROUP BY group_columns
HAVING aggregate_condition
ORDER BY columns;
```

As an example, the query in Listing 5.3 uses a HAVING condition.

Listing 5.3: Using the HAVING condition

```
SELECT launch_dt,
  MAX(price) MAX ,
  MIN(price) MIN ,
  SUM(price) SUM ,
  AVG(price) AVG ,
  COUNT(price) CNT ,
  COUNT(*) AS 'C(*)'
FROM product
GROUP BY price,
  launch_dt
HAVING COUNT(price) > 1
ORDER BY price,
  launch_dt;
```

Only groups having more than one row (satisfying the COUNT(price) > 1 condition) will be returned. Only one row will be returned, the one with **price** = 15 and **launch_dt** = 29-MAR-13.

```
+------------+-------+-------+-------+-----------+-----+------+
| launch_dt  | MAX   | MIN   | SUM   | AVG       | CNT | C(*) |
+------------+-------+-------+-------+-----------+-----+------+
| 2013-03-29 | 15.00 | 15.00 | 30.00 | 15.000000 |   2 |    2 |
+------------+-------+-------+-------+-----------+-----+------+
```

If a WHERE clause is present, it must appear after the GROUP BY clause. Individual rows will be selected by the WHERE condition first before grouping occurs. For instance, the query in Listing 5.4 uses both WHERE and GROUP BY.

Listing 5.4: Grouping with WHERE

```
SELECT launch_dt,
  MAX(price) MAX ,
  MIN(price) MIN ,
  SUM(price) SUM ,
  AVG(price) AVG ,
  COUNT(price) CNT ,
  COUNT(*) AS 'C(*)'
FROM product
WHERE p_name NOT LIKE 'Super%'
GROUP BY launch_dt
HAVING launch_dt > '2013-03-29'
ORDER BY launch_dt;
```

Here is the query output.

```
+------------+-------+-------+-------+-----------+-----+------+
| launch_dt  | MAX   | MIN   | SUM   | AVG       | CNT | C(*) |
+------------+-------+-------+-------+-----------+-----+------+
| 2013-03-30 | 25.00 | 25.00 | 25.00 | 25.000000 |   1 |    1 |
| 2013-03-31 | 10.00 | 10.00 | 10.00 | 10.000000 |   1 |    1 |
+------------+-------+-------+-------+-----------+-----+------+
```

In this case, Super_Nut does not satisfy the WHERE condition. As such, it is not included in the aggregation.

Applying aggregate as a WHERE condition clause is not allowed. This is shown in Listing 5.5, which contains a query that throws an error if executed.

Listing 5.5: Error with WHERE on the aggregate

```
SELECT price,
  launch_dt,
  MAX(price) MAX,
  MIN(price) MIN,
  SUM(price) SUM,
  AVG(price) AVG,
  COUNT(price) CNT,
  COUNT(*) "C(*)"
FROM product
WHERE COUNT(price) > 1;
```

Executing this query will give you this error message.

```
ERROR 1111 (HY000): Invalid use of group function
```

Summary

In this chapter you learned how to aggregate values from rows. You also learned to use the HAVING condition applied on aggregates. In the next chapter you will learn about the JOIN clause used to “aggregate” rows from more than one table.

Chapter 6
Joins

A real-world database typically stores data in dozens or even hundreds of tables. In these multi-table databases, a table often relates to one or some other tables. In this environment, you should be able to relate rows from two or more tables by using the JOIN clause. This chapter shows you how.

Primary Keys and Foreign Keys

In Chapter 1, “Storing and Maintaining Data” you learned about the primary key. A primary key is a column, or a set of columns, which uniquely identifies every row in a table. A foreign key is a column, or a set of columns, which is used to relate to the primary key of another table. The process of using the foreign key/primary key to relate rows from two tables is called joining.

While a primary key must be unique, a foreign key does not have to be. You can have a foreign key in more than one row. For example, in a **customer_order** table you can have many orders for the same product. In this **customer_order** table, a product is represented by its foreign key, e.g. **product_code**, which is the primary key of the **product** table.

Even though the use of primary and foreign keys is not an absolute requirement for joining tables, their absence may cause you to incorrectly join tables.

Querying Multiple Tables

To query data from multiple tables, use the JOIN keyword to specify the related columns from two tables. The JOIN clause of a SELECT statement joins related rows from two or more tables, based on their primary key/foreign key relationship.

For example, a **customer_order** (**c_order**) table may need a foreign key column to relate to the primary key of the **product** table. Additionally, the **customer_order** table may also need a foreign key to relate to the primary key of the **customer** table.

The syntax for the JOIN is as follows.

```
SELECT columns FROM table_1, table_2, ... table_n
WHERE table_1.primary_key = table_2.foreign_key
AND table_2.primary_key = table_n.foregin_key;
```

To illustrate the use of joins, I will use the **c_order**, **customer_table**, and **product** tables in Table 6.1, Table 6.2, and Table 6.3, respectively. The **C_NO** and **P_CODE** columns in the **c_order** table are foreign keys; their related primary keys reside in the **customer** and **product** tables, respectively.

```
+------+--------+------+------------+
| C_NO | P_CODE | QTY  | ORDER_DT   |
+------+--------+------+------------+
| 10   | 1      |  100 | 2013-04-01 |
| 10   | 2      |  100 | 2013-04-01 |
| 20   | 1      |  200 | 2013-04-01 |
| 30   | 3      |  300 | 2013-04-02 |
| 40   | 4      |  400 | 2013-04-02 |
| 40   | 5      |  400 | 2013-04-03 |
+------+--------+------+------------+
```

Table 6.1: The customer order (c_order) table

```
+------+----------------+
| c_no | c_name         |
+------+----------------+
| 10   | Standard Store |
| 20   | Quality Store  |
| 30   | Head Office    |
| 40   | Super Agent    |
+------+----------------+
```

Table 6.2: The customer table

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 1      | Nail      | 10.00 | 2013-03-31 |
| 2      | Washer    | 15.00 | 2013-03-29 |
| 3      | Nut       | 15.00 | 2013-03-29 |
| 4      | Screw     | 25.00 | 2013-03-30 |
| 5      | Super_Nut | 30.00 | 2013-03-30 |
| 6      | New Nut   |  NULL | NULL       |
+--------+-----------+-------+------------+
```

Table 6.3: The product table

Listing 6.1 is an example of a JOIN query. It joins the rows from the **c_order** table to the rows from the **customer** table based on the **c_no** foreign key column of the **c_order** table and the **c_no** primary key column of the **customer** table. The query returns the name of every customer who has placed one or more orders.

Listing 6.1: A two table join

```
SELECT c_name,
  p_code,
  c_order.qty,
  c_order.order_dt
FROM c_order
JOIN customer
ON c_order.c_no = customer.c_no;
```

Applied against the example **c_order** and **customer** tables, the query result is as follows.

```
+----------------+--------+------+------------+
| c_name         | p_code | qty  | order_dt   |
+----------------+--------+------+------------+
| Standard Store | 1      |  100 | 2013-04-01 |
| Standard Store | 2      |  100 | 2013-04-01 |
| Quality Store  | 1      |  200 | 2013-04-01 |
| Head Office    | 3      |  300 | 2013-04-02 |
| Super Agent    | 4      |  400 | 2013-04-02 |
| Super Agent    | 5      |  400 | 2013-04-03 |
+----------------+--------+------+------------+
```

Using Table Aliases

In a join query, different tables can have columns with identical names. To make sure you refer to the correct column of a table, you need to qualify it with its table. In the previous example, **c_order.c_no** (the **c_no** column of the **c_order** table) and **customer.c_no** (the **c_no** column of the **customer** table) were how the **c_no** columns were qualified. A table alias can be a more convenient (and shorter) way to qualify a column.

For example, in the query in Listing 6. 2, **o** is an alias for the **c_order** table and **c** is an alias for the **customer** table. These aliases are then used in the ON clause to qualify the **c_no** columns with their respective tables.

Listing 6.2: Using table aliases

```
SELECT c_name,
  p_code,
  o.qty,
  o.order_dt
FROM c_order o
JOIN customer c
ON o.c_no = c.c_no;
```

Column Aliases vs. Table Aliases

In Chapter 4, “Query Output”, I explained the use of aliases for columns using the AS keyword. Although a column alias can be created without using the AS keyword, its presence improves readability (“p_name AS product_name” instead of “p_name product_name”). On the other hand, table aliases cannot use the AS keyword.

Joining More than Two Tables

Using the JOIN syntax presented earlier, you can join more than two tables. To do this, in the SELECT statement, join two tables at a time.

For example, the query in Listing 6.3 joins the **c_order** table to the **customer** table, and then joins the **customer** table to the **product** table. The rows in the **c_order** table are joined to the rows of the same **c_no** column from the **customer** table, and these rows are then joined to the rows with the same **p_code** from the **product** table. This query returns customer names and their orders.

Listing 6.3: A three table join

```
SELECT c_name,
  p_name,
  o.qty,
  o.order_dt
FROM c_order o
JOIN customer c
ON o.c_no = c.c_no
JOIN product p
ON o.p_code = p.p_code;
```

Applied against the **c_order**, **customer** and **product** sample tables, you will see the following result.

```
+----------------+-----------+------+------------+
| c_name         | p_name    | qty  | order_dt   |
+----------------+-----------+------+------------+
| Standard Store | Nail      |  100 | 2013-04-01 |
| Standard Store | Washer    |  100 | 2013-04-01 |
| Quality Store  | Nail      |  200 | 2013-04-01 |
| Head Office    | Nut       |  300 | 2013-04-02 |
| Super Agent    | Screw     |  400 | 2013-04-02 |
| Super Agent    | Super_Nut |  400 | 2013-04-03 |
+----------------+-----------+------+------------+
```

You can also apply WHERE conditions for selecting rows on a join query. For example, in Listing 6.4, thanks to the WHERE condition, only products with names that do not start with “Super” will be in the query output.

Listing 6.4: JOIN and WHERE

```
SELECT c_name,
  p_name,
  o.qty,
  o.order_dt
FROM c_order o
JOIN customer c
ON o.c_no = c.c_no
JOIN product p
ON o.p_code = p.p_code
WHERE p_name NOT LIKE 'Super%';
```

Executing the query in Listing 6.4 against the sample tables will produce the following output rows.

```
+----------------+--------+------+------------+
| c_name         | p_name | qty  | order_dt   |
+----------------+--------+------+------------+
| Standard Store | Nail   |  100 | 2013-04-01 |
| Standard Store | Washer |  100 | 2013-04-01 |
| Quality Store  | Nail   |  200 | 2013-04-01 |
| Head Office    | Nut    |  300 | 2013-04-02 |
| Super Agent    | Screw  |  400 | 2013-04-02 |
+----------------+--------+------+------------+
```

Joining on More than One Column

The preceding joins were on one column. Tables can also be joined on more than one column.

The syntax for a multicolumn join for two tables is as follows.

```
SELECT columns FROM table_1, table_2
WHERE table_1.column_1 = table_2.column_1
AND table_1.column_2 = table_2.column_2
...
AND table_1.column_n = table_2.column_n;
```

As an example, suppose you want to track order shipments in the following **shipment** table

```
+------+--------+------------+----------+------------+
| C_NO | P_CODE | ORDER_DT   | SHIP_QTY | SHIP_DT    |
+------+--------+------------+----------+------------+
| 10   | 1      | 2013-04-01 |       50 | 2013-04-02 |
| 10   | 2      | 2013-04-01 |      100 | 2013-04-02 |
| 20   | 1      | 2013-04-01 |      100 | 2013-04-02 |
| 30   | 3      | 2013-04-02 |      300 | 2013-04-03 |
| 10   | 1      | 2013-04-01 |       50 | 2013-04-10 |
+------+--------+------------+----------+------------+
```

To retrieve the order quantity (the **qty** column of the **c_order** table) of each shipment, you need to have a query that joins the **shipment** table to the **order** table on three columns, **c_no**, **p_no**, and **order_dt**, as shown in the query in Listing 6.5.

Listing 6.5: A multiple columns join

```
SELECT o.c_no,
  o.p_code,
  o.order_dt,
  ship_qty,
  ship_dt,
  qty
FROM shipment s
JOIN c_order o
ON s.c_no      = o.c_no
AND s.p_code   = o.p_code
AND s.order_dt = o.order_dt;
```

Executing this query against the **c_order** and **shipment** tables will give you the following output rows.

```
+------+--------+------------+----------+------------+------+
| c_no | p_code | order_dt   | ship_qty | ship_dt    | qty  |
+------+--------+------------+----------+------------+------+
| 10   | 1      | 2013-04-01 |       50 | 2013-04-02 |  100 |
| 10   | 1      | 2013-04-01 |       50 | 2013-04-10 |  100 |
| 10   | 2      | 2013-04-01 |      100 | 2013-04-02 |  100 |
| 20   | 1      | 2013-04-01 |      100 | 2013-04-02 |  200 |
| 30   | 3      | 2013-04-02 |      300 | 2013-04-03 |  300 |
+------+--------+------------+----------+------------+------+
```

The Outer Join

All the joins I explained so far were inner joins. There is another type of join, the outer join. While an inner join query produces only related rows from the joined tables, an outer join query produces all rows from one table even when some of the rows do not have matching rows from the other table.

There are two subtypes of outer joins, LEFT and RIGHT. The following points describes each of these two types.

All rows from the table on the left of the left outer join will be in the output whether or not there are matching rows from the table on its right. The syntax for the left outer join is as follows.

```
SELECT columns
FROM table_1 LEFT OUTER JOIN table_2
ON table_1.column = table_2.column ... ;
```

All rows form the table on the right of the right outer join will be in the output whether or not there are matching rows from the table on its left. The syntax for the right outer join is as follows.

```
SELECT columns FROM table_1 RIGHT OUTER JOIN table_2 ON
      table_1.column = table_2.column ... ;
```

Listing 6.6 is an example left outer join query. This query returns all rows from the **c_order** table.

Listing 6.6: Left outer join

```
SELECT o.*,
  ship_dt
FROM c_order o
LEFT OUTER JOIN shipment s
ON o.p_code = s.p_code
AND o.c_no  = s.c_no;
```

If you run this query against our example **c_order** and **shipment** tables, you will see the following output rows.

```
+------+--------+------+------------+------------+
| C_NO | P_CODE | QTY  | ORDER_DT   | ship_dt    |
+------+--------+------+------------+------------+
| 10   | 1      |  100 | 2013-04-01 | 2013-04-02 |
| 10   | 2      |  100 | 2013-04-01 | 2013-04-02 |
| 20   | 1      |  200 | 2013-04-01 | 2013-04-02 |
| 30   | 3      |  300 | 2013-04-02 | 2013-04-03 |
| 10   | 1      |  100 | 2013-04-01 | 2013-04-10 |
| 40   | 4      |  400 | 2013-04-02 | NULL       |
| 40   | 5      |  400 | 2013-04-03 | NULL       |
+------+--------+------+------------+------------+
```

Note that the last two rows have no matching rows from the shipment **table** and consequently their **ship_dt** column has NULL values.

Rows with NULL Only

If you want to query only orders that have not been shipped at all, you have to put this “only” condition in the WHERE clause of your query (ship_dt IS NULL) as in the query in Listing 6.7.

Listing 6.7: NULL only rows

```
SELECT o.*,
  ship_dt
FROM c_order o
LEFT OUTER JOIN shipment s
ON o.p_code = s.p_code
AND o.c_no  = s.c_no
WHERE s.ship_dt IS NULL;
```

The following output rows from the query in Listing 6.17 are customer orders that have not been shipped.

```
+------+--------+------+------------+---------+
| C_NO | P_CODE | QTY  | ORDER_DT   | ship_dt |
+------+--------+------+------------+---------+
| 40   | 4      |  400 | 2013-04-02 | NULL    |
| 40   | 5      |  400 | 2013-04-03 | NULL    |
+------+--------+------+------------+---------+
```

Self-Joins

Assuming some of your products have substitutes and you want to record the substitutes in the **product** table, you then need to add a column. The new column, which is called **s_code** in the **product** table, contains the product code of the substitute.

The new product table, with a row having s_code 5, now looks like the following.

```
+--------+-----------+-------+------------+--------+
| p_code | p_name    | price | launch_dt  | s_code |
+--------+-----------+-------+------------+--------+
| 1      | Nail      | 10.00 | 2013-03-31 | NULL   |
| 2      | Washer    | 15.00 | 2013-03-29 | NULL   |
| 3      | Nut       | 15.00 | 2013-03-29 | 5      |
| 4      | Screw     | 25.00 | 2013-03-30 | NULL   |
| 5      | Super_Nut | 30.00 | 2013-03-30 | NULL   |
| 6      | New Nut   |  NULL | NULL       | NULL   |
+--------+-----------+-------+------------+--------+
```

To add the **s_code** column, execute the following statement:

```
ALTER TABLE product ADD (s_code VARCHAR2(6));
```

Then, to update the p_code = 3 row, execute the following statement:

```
UPDATE product SET s_code = 5 WHERE p_code = 3;
```

If you need to know the product name of a substitute, you need the query shown in Listing 6.8. This query joins the product table to itself. This kind of join is called a self-join.

The syntax for the self join is as follows.

```
SELECT columns
FROM table alias_1
JOIN table alias_2
ON alias_1.column_x = alias_2.column_y;
```

Note that *column_x* and *column_y* are columns in the same table.

Listing 6.8: A self-join

```
SELECT prod.p_code,
  prod.p_name,
  subst.p_code subst_p_code,
  subst.p_name subst_name
FROM product prod
LEFT OUTER JOIN product subst
ON prod.s_code = subst.p_code
ORDER BY prod.p_code;
```

Here are the output rows of the query, showing “Newer Nut” in the **subst_name** column of the third row.

```
+--------+-----------+--------------+------------+
| p_code | p_name    | subst_p_code | subst_name |
+--------+-----------+--------------+------------+
| 1      | Nail      | NULL         | NULL       |
| 2      | Washer    | NULL         | NULL       |
| 3      | Nut       | 5            | Super_Nut  |
| 4      | Screw     | NULL         | NULL       |
| 5      | Super_Nut | NULL         | NULL       |
| 6      | New Nut   | NULL         | NULL       |
+--------+-----------+--------------+------------+
```

Multiple Uses of A Table

If a product can have more than one substitute, you need to store the product-substitute relationships in a separate table. A substitute cannot be recorded in the **product** table.

To create the table that stores the product-substitute relationships named **prod_subst**, execute the following statement.

```
CREATE TABLE prod_subst (p_code VARCHAR2(6), s_code VARCHAR2(6));
```

To remove the **s_code** column, execute the following statement:

```
ALTER TABLE product DROP (s_code);
```

Your **product** table will now contain the following rows.

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 1      | Nail      | 10.00 | 2013-03-31 |
| 2      | Washer    | 15.00 | 2013-03-29 |
| 3      | Nut       | 15.00 | 2013-03-29 |
| 4      | Screw     | 25.00 | 2013-03-30 |
| 5      | Super_Nut | 30.00 | 2013-03-30 |
| 6      | New Nut   |  NULL | NULL       |
+--------+-----------+-------+------------+
```

Assuming that the only product with substitutes is product number 3 and its substitutes are the products number 5 and 6, the **prod_subst** table will have the following two rows. (You need to insert these two rows using INSERT statements).

```
+--------+--------+
| p_code | s_code |
+--------+--------+
| 3      | 5      |
| 3      | 6      |
+--------+--------+
```

To get the name of a product and the names of its substitutes, you need to use the **product** table twice, as shown in the query in Listing 6.9.

Listing 6.9: Multiple uses of a table

```
SELECT prod.p_code,
  prod.p_name,
  ps.s_code,
  subst.p_name AS s_name
FROM product prod
INNER JOIN prod_subst ps
ON prod.p_code = ps.p_code
INNER JOIN product subst
ON ps.s_code = subst.p_code
ORDER BY prod.p_code;
```

Here are the output rows from the query in Listing 6.9.

```
+--------+--------+--------+-----------+
| p_code | p_name | s_code | s_name    |
+--------+--------+--------+-----------+
| 3      | Nut    | 5      | Super_Nut |
| 3      | Nut    | 6      | New Nut   |
+--------+--------+--------+-----------+
```

Natural Joins

If two tables have columns that share a name, you can naturally join the two tables on these columns. In a natural join, you do not need to specify the columns that the join should use.

The syntax for the natural join is this.

```
SELECT columns FROM table_1 NATURAL JOIN table_2 ... ;
```

Listing 6.10 shows an example of a natural join on the **c_order** and **customer** tables. This natural join implicitly joins the tables on their **c_no** columns.

Listing 6.10: A natural join

```
SELECT * FROM c_order NATURAL JOIN customer;
```

Running the query in Listing 6.10 gives you the following output rows.

```
+------+--------+------+------------+----------------+
| C_NO | P_CODE | QTY  | ORDER_DT   | c_name         |
+------+--------+------+------------+----------------+
| 10   | 1      |  100 | 2013-04-01 | Standard Store |
| 10   | 2      |  100 | 2013-04-01 | Standard Store |
| 20   | 1      |  200 | 2013-04-01 | Quality Store  |
| 40   | 4      |  400 | 2013-04-02 | Super Agent    |
| 40   | 5      |  400 | 2013-04-03 | Super Agent    |
+------+--------+------+------------+----------------+
```

Natural Outer Joins

The natural join is also applicable to the outer join. Consider the query in Listing 6.11.

Listing 6.11: A natural outer join

```
SELECT * FROM c_order NATURAL RIGHT JOIN customer;
```

Applying the query against the **c_order** and **customer** tables will give you the following output rows.

```
+------+----------------+--------+------+------------+
| c_no | c_name         | p_code | qty  | order_dt   |
+------+----------------+--------+------+------------+
| 10   | Standard Store | 1      |  100 | 2013-04-01 |
| 10   | Standard Store | 2      |  100 | 2013-04-01 |
| 20   | Quality Store  | 1      |  200 | 2013-04-01 |
| 30   | Head Office    | NULL   | NULL | NULL       |
| 40   | Super Agent    | 4      |  400 | 2013-04-02 |
| 40   | Super Agent    | 5      |  400 | 2013-04-03 |
+------+----------------+--------+------+------------+
```

Mixing Natural Joins with Different Column Names

If you need to join on more than one column, and the second column pair does not share a name, you can specify the different column names in the WHERE clause. Listing 6.12 shows an example of such a case.

Listing 6.12: Mixing natural join with different column names

```
SELECT * FROM c_order o NATURAL RIGHT JOIN product p WHERE
      o.order_dt = p.launch_dt;
```

In the query in Listing 6.12, in addition to the natural join on the **c_no** column, the rows from the two tables have to be joined on the two dates.

The query does not return any row as we don’t have any order of a product with the same order date as the product’s launch date.

The USING Keyword

A natural join will use all columns with the same names from the joined tables. If you want your query to join only on some of these identically named columns, instead of using the NATURAL keyword, use the **USING keyword**.

The syntax for joining two tables with USING is as follows.

```
SELECT columns
FROM table_1
JOIN table_2 USING (column);
```

Listing 6.13, for example, joins the **c_order** table to the **shipment** table only on their **p_code** columns. It does not join the tables on their **c_no** columns. This query gives you the total quantity shipped by product code.

Listing 6.13: Using USING

```
SELECT p_code,
  SUM(s.ship_qty)
FROM c_order o
JOIN shipment s USING (p_code)
GROUP BY p_code;
```

Executing this query against our example **c_order** and **shipment** tables will produce the following output rows.

```
+--------+-----------------+
| p_code | SUM(s.ship_qty) |
+--------+-----------------+
| 1      |             400 |
| 2      |             100 |
| 3      |             300 |
+--------+-----------------+
```

Summary

In this chapter you learned about getting data from multiple tables. You learned how to use the various types of joins for this purpose.

Chapter 7
Subqueries

A subquery is a query nested within another query. The containing query is called an outer query. A subquery in turn can have a nested query, making it a multiple nested query.

This chapter discusses subqueries in detail.

Single-Row Subqueries

A single-row subquery is a subquery that returns a single value. A single-row subquery can be placed in the WHERE clause of an outer query. The return value of the subquery is compared with a column of the outer query using one of the comparison operators. (Comparison operators were discussed in Chapter 2, “Basic Queries”)

For example, the query in Listing 7.1 contains a single-row subquery that returns the highest sale price recorded for a product. The outer query returns all products from the **product** table that have that highest price (30.00), the Super_Nut and Newer Nut products.

Listing 7.1: A subquery that returns a single value

```
SELECT *
FROM product
WHERE price =
  (SELECT MAX(price)
  FROM product p
  INNER JOIN c_order o
  ON p.p_code = o.p_code
  );
```

Note that the subquery in Listing 7.1 is printed in bold.

Executing the query in Listing 7.1 against this **product** table

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 1      | Nail      | 10.00 | 2013-03-31 |
| 2      | Washer    | 15.00 | 2013-03-29 |
| 3      | Nut       | 15.00 | 2013-03-29 |
| 4      | Screw     | 25.00 | 2013-03-30 |
| 5      | Super_Nut | 30.00 | 2013-03-30 |
| 6      | New Nut   | 30.00 | 2013-05-01 |
+--------+-----------+-------+------------+
```

and the following **c_order** (customer order) table

```
+------+--------+------+------------+
| C_NO | P_CODE | QTY  | ORDER_DT   |
+------+--------+------+------------+
| 10   | 1      |  100 | 2013-04-01 |
| 10   | 2      |  100 | 2013-04-01 |
| 20   | 1      |  200 | 2013-04-01 |
| 30   | 3      |  300 | 2013-04-02 |
| 40   | 4      |  400 | 2013-04-02 |
| 40   | 5      |  400 | 2013-04-03 |
+------+--------+------+------------+
```

will give you the following result.

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 5      | Super_Nut | 30.00 | 2013-03-30 |
| 6      | New Nut   | 30.00 | 2013-05-01 |
+--------+-----------+-------+------------+
```

The column and subquery result do not have to be the same column, but they must have compatible data types. In the query in Listing 7.1, the price column of the product table is a numeric type and the subquery also returns a numeric type.

If the subquery returns more than one value, you will get an error message. For example, the query in Listing 7.2 throws an error because the subquery returns more than one value.

Listing 7.2: Single-row subquery error

```
SELECT *
FROM product
WHERE price =
  (SELECT MAX(price)
  FROM product p
  INNER JOIN c_order s
  ON p.p_code = s.p_code
  GROUP BY p.launch_dt
  );
```

Here is the error that you will see if you run the query in Listing 7.2.

```
ERROR 1242 (21000): Subquery returns more than 1 row
```

Multiple-Row Subqueries

A subquery that returns more than one value is called a multiple-row subquery. This type of subquery also occurs in the WHERE clause of an outer query. However, instead of using a comparison operator, you use IN or NOT IN in the WHERE clause of a multiple-row subquery.

For example, the query in Listing 7.3 contains a multiple-row subquery.

Listing 7.3: Using a multiple-row subquery

```
SELECT *
FROM product
WHERE price IN
  (SELECT MAX(price)
  FROM product p
  INNER JOIN c_order s
  ON p.p_code = s.p_code
  GROUP BY p.launch_dt
  );
```

Run against the same product and order tables, the subquery will return these three values:

```
+------------+
| MAX(price) |
+------------+
|      15.00 |
|      30.00 |
|      10.00 |
+------------+
```

The overall query output will be as follows.

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 1      | Nail      | 10.00 | 2013-03-31 |
| 2      | Washer    | 15.00 | 2013-03-29 |
| 3      | Nut       | 15.00 | 2013-03-29 |
| 5      | Super_Nut | 30.00 | 2013-03-30 |
| 6      | New Nut   | 30.00 | 2013-05-01 |
+--------+-----------+-------+------------+
```

The ALL and ANY Operators

In addition to IN and NOT IN, you can also use the ALL or ANY operators in a multiple-row subquery. With ALL or ANY you use a comparison operator. For instance, the query in Listing 7.4 employs the ALL operator to compare the price column with the subquery result.

Listing 7.4: Using ALL

```
SELECT *
FROM product
WHERE price >= ALL
  (SELECT MAX(price)
  FROM product p
  INNER JOIN c_order o
  ON p.p_code = o.p_code
  GROUP BY p.launch_dt
  )
ORDER BY p_code;
```

Run against the same **product** and **c_order** tables, the subquery in Listing 7.4 (printed in bold) returns these results:

```
+------------+
| MAX(price) |
+------------+
|      15.00 |
|      30.00 |
|      10.00 |
+------------+
```

The query output will consist of only rows whose price is greater or equal to all the values returned by the subquery.

Here is the query output.

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 5      | Super_Nut | 30.00 | 2013-03-30 |
| 6      | New Nut   | 30.00 | 2013-05-01 |
+--------+-----------+-------+------------+
```

As another example, the query in Listing 7.5 is similar to the one in Listing 7.4 except that the equal operator is used to compare the price with the subquery result.

Listing 7.5: Using ALL for equal comparison

```
SELECT *
FROM product
WHERE price = ALL
  (SELECT MAX(price)
  FROM product p
  INNER JOIN c_order o
  ON p.p_code = o.p_code
  GROUP BY p.launch_dt
  )
ORDER BY p_code;
```

As in the previous example, the subquery will return these values.

```
+------------+
| MAX(price) |
+------------+
|      15.00 |
|      30.00 |
|      10.00 |
+------------+
```

The query output consists of only rows whose price equals to all these values, which is not possible as each product has only one price. As a result, the query returns no rows. Here is a message you will see if you run the query.

```
Empty set (0.00 sec)
```

Subqueries Returning Rows Having the Same Value

If you use ALL and the subquery returns one row or multiple rows with the same value you will not get a “no rows selected” message.

You can also use the ANY operator to compare a column with the values returned by a subquery. If you use ALL, the query compares a column to all values (every one of the values) returned by the subquery. If you use ANY, the query compares a column to any one of the values returned by the subquery.

For example, the query in Listing 7.6 compares the price column for equality to any of the maximum prices returned by the subquery, in other words, the WHERE condition is true if the price equals to any of maximum prices.

Listing 7.6: Using the equal comparison to ANY value

```
SELECT *
FROM product
WHERE price = ANY
  (SELECT MAX(price)
  FROM product p
  INNER JOIN c_order o
  ON p.P_code = o.p_code
  GROUP BY p.launch_dt
  )
ORDER BY p_code;
```

The subquery will return these rows.

```
+------------+
| MAX(price) |
+------------+
|      15.00 |
|      30.00 |
|      10.00 |
+------------+
```

The outer query output will consist of any product that has a price equal to any of these (maximum price) values. Here is the query output.

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 1      | Nail      | 10.00 | 2013-03-31 |
| 2      | Washer    | 15.00 | 2013-03-29 |
| 3      | Nut       | 15.00 | 2013-03-29 |
| 5      | Super_Nut | 30.00 | 2013-03-30 |
| 6      | New Nut   | 30.00 | 2013-05-01 |
+--------+-----------+-------+------------+
```

Multiple Nested Subqueries

A subquery can contain another query, making it a query with multiple nested subqueries. The query in Listing 7.7, for example, has multiple nested subqueries. Notice the two IN's, one for each of the two nested queries? The query returns only customers who have not ordered any product having name that contains 'Nut'.

Listing 7.7: Query with multiple nested subqueries

```
SELECT customer.*
FROM customer
WHERE c_no IN
  (SELECT c_no
  FROM c_order
  WHERE p_code IN
    (SELECT p_code FROM product WHERE p_name NOT LIKE '%Nut%'
    )
  );
```

Here is the query result.

```
+------+----------------+
| c_no | c_name         |
+------+----------------+
| 10   | Standard Store |
| 20   | Quality Store  |
| 40   | Super Agent    |
+------+----------------+
```

Correlated Subqueries

All the preceding subqueries are independent of their outer queries. A subquery can also be related to its outer query, where one or more column from the outer query table is (are) related to the column(s) of the subquery table in the WHERE clause of the subquery. This type of subquery is called the correlated subquery.

As an example, the query in Listing 7.8 contains a correlated subquery that returns only customers who have not ordered any product whose name contains ‘Nut’. Note that the **c_no** column of the outer query table, **customer**, is related to the **c_no** column of the **c_order** table of the subquery.

Listing 7.8: Using a correlated subquery

```
SELECT customer.*
FROM customer
WHERE c_no IN
  (SELECT c_no
  FROM c_order o
  JOIN product p
  ON o.p_code = p.p_code
  WHERE p_name NOT LIKE '%Nut%'
  AND customer.c_no = o.c_no
  );
```

The following are the query result.

```
+------+----------------+
| c_no | c_name         |
+------+----------------+
| 10   | Standard Store |
| 20   | Quality Store  |
| 40   | Super Agent    |
+------+----------------+
```

Summary

In this chapter you learned the various types of subqueries, such as nested and correlated subqueries. In the next chapters you will apply the lesson you learned in this chapter to combine the results of two or more queries.

Chapter 8
Compound Queries

You can combine the results of two or more SELECT statements using the UNION ALL or UNION operators. The number of output columns from every statement must be the same and the corresponding columns must have identical or compatible data types.

This chapter shows you how to combine query results.

UNION ALL

When you combine two or more queries with the UNION ALL operator, the overall output will be the rows from all the queries. For example, take a look at the query in Listing 8.1. This query consists of two SELECT statements.

Listing 8.1: Using UNION ALL

```
SELECT p_code, p_name, 'FIRST QUERY' query
FROM product p WHERE p_name LIKE '%Nut%'
UNION ALL
SELECT p.p_code,
  p_name,
  'SECOND_QUERY' query
FROM c_order o
INNER JOIN product p
ON o.p_code = p.p_code;
```

Note that the 'FIRST QUERY' and 'SECOND_QUERY' literals in the first and second SELECT statements, respectively, are just labels to identify where a row is coming from.

Assuming that the **product** table has the following rows

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 1      | Nail      | 10.00 | 2013-03-31 |
| 2      | Washer    | 15.00 | 2013-03-31 |
| 3      | Nut       | 15.00 | 2013-03-29 |
| 4      | Screw     | 25.00 | 2013-03-30 |
| 5      | Super_Nut | 30.00 | 2013-03-30 |
| 6      | New Nut   |  NULL | NULL       |
+--------+-----------+-------+------------+
```

and the **c_order** table contains the following records

```
+------+--------+------+------------+
| C_NO | P_CODE | QTY  | ORDER_DT   |
+------+--------+------+------------+
| 10   | 1      |  100 | 2013-04-01 |
| 10   | 2      |  100 | 2013-04-01 |
| 20   | 1      |  200 | 2013-04-01 |
| 30   | 3      |  300 | 2013-04-02 |
| 40   | 4      |  400 | 2013-04-02 |
| 40   | 5      |  400 | 2013-04-03 |
+------+--------+------+------------+
```

the query in Listing 8.1 will return the following output.

```
+--------+-----------+--------------+
| p_code | p_name    | query        |
+--------+-----------+--------------+
| 3      | Nut       | FIRST QUERY  |
| 5      | Super_Nut | FIRST QUERY  |
| 6      | New Nut   | FIRST QUERY  |
| 1      | Nail      | SECOND_QUERY |
| 2      | Washer    | SECOND_QUERY |
| 1      | Nail      | SECOND_QUERY |
| 3      | Nut       | SECOND_QUERY |
| 4      | Screw     | SECOND_QUERY |
| 5      | Super_Nut | SECOND_QUERY |
+--------+-----------+--------------+
```

Note that the output of the query in Listing 8.1 comprises all the records form the first SELECT statement followed by the rows from the second SELECT statement. You can of course use the ORDER BY clause to re-order this. For instance, the query in Listing 8.2 modifies the query in

Listing 8.1 by ordering the results on the **p_code** column using the ORDER BY clause.

Listing 8.2: Ordering output rows of a compound query

```
SELECT p_code, p_name, 'FIRST QUERY' query
FROM product p WHERE p_name LIKE '%Nut%'
UNION ALL
SELECT p.p_code,
  p_name,
  'SECOND_QUERY' query
FROM c_order o
INNER JOIN product p
ON o.p_code = p.p_code
ORDER BY p_code;
```

The result of the query in Listing 8.2 is as follows.

```
+--------+-----------+--------------+
| p_code | p_name    | query        |
+--------+-----------+--------------+
| 1      | Nail      | SECOND_QUERY |
| 1      | Nail      | SECOND_QUERY |
| 2      | Washer    | SECOND_QUERY |
| 3      | Nut       | SECOND_QUERY |
| 3      | Nut       | FIRST QUERY  |
| 4      | Screw     | SECOND_QUERY |
| 5      | Super_Nut | FIRST QUERY  |
| 5      | Super_Nut | SECOND_QUERY |
| 6      | New Nut   | FIRST QUERY  |
+--------+-----------+--------------+
```

UNION

UNION is similar to UNION ALL. However, with UNION duplicate rows will be returned once only. For example, consider the query in Listing 8.3 that consists of two SELECT elements.

Listing 8.3: Using UNION

```
SELECT p_code,
  p_name
FROM product p
WHERE p_name LIKE '%Nut%'
```

```
UNION
SELECT p.p_code,
  p_name
FROM c_order o
INNER JOIN product p
ON o.p_code = p.p_code
ORDER BY p_code;
```

Here is the output of the query.

```
+--------+-----------+
| p_code | p_name    |
+--------+-----------+
| 1      | Nail      |
| 2      | Washer    |
| 3      | Nut       |
| 4      | Screw     |
| 5      | Super_Nut |
| 6      | New Nut   |
+--------+-----------+
```

Summary

In this chapter you learned to combine the output of two or more SELECT statements. There are five operators you can use for this purpose, UNION ALL, UNION, INTERSECT, and MINUS.

Chapter 9 Views

A view is effectively a predefined query. You create and use a view most frequently for the following purposes:

- Hiding table columns (for security protection)
- Presenting pre-computed columns (in lieu of table columns)
- Hiding queries (so that the query outputs are available without running the queries)

This chapter discusses the view and presents examples of views.

Creating and Using Views

You create a view using the CREATE VIEW statement. Here is its syntax.

```
CREATE VIEW view_name (columns) AS SELECT ... ;
```

The SELECT statement at the end of the CREATE VIEW statement is a predefined query. When you use a view its predefined query is executed. Since a query result is a table that is not persisted (stored) in the database, a view is also known as a virtual table. The table in the SELECT statement of a view is known as a base table.

One of the reasons you use a view is when you have a table you need to share with other people. If you don't want some of the table columns viewable by others, you can use a view to hide those columns. You would then share the view and restrict access to the base table.

For example, Listing 9.1 shows an SQL statement for creating a view called **product_v** that is based on the **product** table. The view hides the price column of the base table.

Listing 9.1: Using a view to hide columns

```
CREATE VIEW product_v
  (p_code , p_name
  ) AS
SELECT p_code, p_name FROM product;
```

The **product_v** view can now be used just as you would any database table. For example, the following statement displays all columns in the **product_v** view.

```
SELECT * FROM product_v WHERE p_name NOT LIKE '%Nut%';
```

Assuming the **product** table contains these rows

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 1      | Nail      | 10.00 | 2013-03-31 |
| 2      | Washer    | 15.00 | 2013-03-29 |
| 3      | Nut       | 15.00 | 2013-03-29 |
| 4      | Screw     | 25.00 | 2013-03-30 |
| 5      | Super_Nut | 30.00 | 2013-03-30 |
| 6      | New Nut   |  NULL | NULL       |
+--------+-----------+-------+------------+
```

the query will return these rows.

```
+--------+--------+
| p_code | p_name |
+--------+--------+
| 1      | Nail   |
| 2      | Washer |
| 4      | Screw  |
+--------+--------+
```

Note that within a database a view name must be unique among all the views and tables in the database.

Another use of the view is to derive computed columns not available in the base table(s). Here is an example.

Suppose the **product** table stores profit margins for each product as follows.

```
+--------+-----------+-------+------------+--------+
| p_code | p_name    | price | launch_dt  | margin |
+--------+-----------+-------+------------+--------+
| 1      | Nail      | 10.00 | 2013-03-31 |      1 |
| 2      | Washer    | 15.00 | 2013-03-29 |      2 |
| 3      | Nut       | 15.00 | 2013-03-29 |      2 |
| 4      | Screw     | 25.00 | 2013-03-30 |      5 |
| 5      | Super_Nut | 30.00 | 2013-03-30 |      5 |
| 6      | New Nut   | 30.00 | 2013-05-01 |      5 |
+--------+-----------+-------+------------+--------+
```

If you want other users to see the selling price (price + margin) but not the supplier price (price) or the margins in the product table, you can create a view that computes the selling price from the product price and margin, as demonstrated by the query in Listing 9.2. This query creates a view called **product_sell_v** that includes a computed column **sell_price**. The value for **sell_price** comes from the price and margin columns in the **product** table.

Listing 9.2: A view with a computed column

```
CREATE VIEW product_sell_v
  (p_no , p_name, sell_price
  ) AS
SELECT p_code, p_name, (price + margin) FROM product;
```

Selecting all data from product_sell_v (using “SELECT * FROM product_sell_v”) returns these rows.

```
+------+-----------+------------+
| p_no | p_name    | sell_price |
+------+-----------+------------+
| 1    | Nail      |      11.00 |
| 2    | Washer    |      17.00 |
| 3    | Nut       |      17.00 |
| 4    | Screw     |      30.00 |
| 5    | Super_Nut |      35.00 |
| 6    | New Nut   |      35.00 |
+------+-----------+------------+
```

Users of a view don’t need to know the details of its predefined query. They only need to know what data is available from the view. Referring back to the self-join example in Chapter 6, “Joins”, you can create the view defined in Listing 9.3 to hide the self-join query. While the rows from the **product** table only have the product code of the substitutes, this view will give you the names of their substitutes as well.

Listing 9.3: Hiding Query

```
CREATE VIEW prod_subs_v AS
SELECT prod.p_code,
  prod.p_name,
  subst.p_code subst_p_code,
  subst.p_name subst_name
FROM product prod
LEFT OUTER JOIN product subst
ON prod.s_code = subst.p_code
ORDER BY prod.p_code;
```

Recall that the **product** table has the following rows.

```
+--------+-----------+-------+------------+--------+--------+
| p_code | p_name    | price | launch_dt  | margin | s_code |
+--------+-----------+-------+------------+--------+--------+
| 1      | Nail      | 10.00 | 2013-03-31 |      1 | 6      |
| 2      | Washer    | 15.00 | 2013-03-29 |      2 | NULL   |
| 3      | Nut       | 15.00 | 2013-03-29 |      2 | 5      |
| 4      | Screw     | 25.00 | 2013-03-30 |      5 | NULL   |
| 5      | Super_Nut | 30.00 | 2013-03-30 |      5 | NULL   |
| 6      | New Nut   | 30.00 | 2013-05-01 |      5 | NULL   |
+--------+-----------+-------+------------+--------+--------+
```

Executing the following query that uses the view created with the statement in Listing 9.3,

```
SELECT * FROM prod_subs_v;
```

produces the following rows.

```
+--------+-----------+--------------+------------+
| p_code | p_name    | subst_p_code | subst_name |
+--------+-----------+--------------+------------+
| 1      | Nail      | 6            | New Nut    |
| 2      | Washer    | NULL         | NULL       |
| 3      | Nut       | 5            | Super_Nut  |
| 4      | Screw     | NULL         | NULL       |
| 5      | Super_Nut | NULL         | NULL       |
| 6      | New Nut   | NULL         | NULL       |
+--------+-----------+--------------+------------+
```

Nested Views

A view can be based on another view. Such a view is called a nested view.

As an example, the **ps_noname_v** view in Listing 9.4 hides the **p_name** column and is based on the **product_sell_v** view created earlier.

Listing 9.4: A nested view

```
CREATE VIEW ps_noname_v
  (p_no , sell_price
  ) AS
SELECT p_no, sell_price FROM product_sell_v;
```

Running this statement

```
SELECT * FROM ps_noname_v;
```

will give you the following output rows

```
+------+------------+
| p_no | sell_price |
+------+------------+
| 1    |      11.00 |
| 2    |      17.00 |
| 3    |      17.00 |
| 4    |      30.00 |
| 5    |      35.00 |
| 6    |      35.00 |
+------+------------+
```

Managing Views

You can easily manage your views in MySQL. To see all views in the current database, execute the following statement.

```
SHOW FULL TABLES WHERE table_type='view';
```

This will return the following output, which may differ from other databases.

```
+-----------------------+------------+
| Tables_in_sales       | Table_type |
+-----------------------+------------+
| prod_subs_v           | VIEW       |
| product_sell_v        | VIEW       |
| product_v             | VIEW       |
| ps_noname_v           | VIEW       |
+-----------------------+------------+
```

To delete a view, use the DROP VIEW statement. The syntax for the DROP VIEW statement is as follows.

```
DROP VIEW view_name;
```

For example, the statement in Listing 9.5 will delete the **ps_noname_v** view.

Listing 9.5: Deleting the ps_noname_v view

```
DROP VIEW ps_noname_v;
```

After running the statement in Listing 9.5, listing the views in the database again will give you these results.

```
+-----------------------+------------+
| Tables_in_sales       | Table_type |
+-----------------------+------------+
| prod_subs_v           | VIEW       |
| product_sell_v        | VIEW       |
| product_v             | VIEW       |
+-----------------------+------------+
```

Summary

A view is a predefined query that you can use to hide columns, include pre-computed columns, and so on. In this chapter you learned how to create and manage views.

Chapter 10
Built-in Functions

The MySQL database provides functions that you can use in your queries. These built-in functions can be grouped into numeric functions, character functions, datetime functions, and functions for handling null values. The objective of this chapter is to introduce you to some of these functions.

Numeric Functions

The following are some of the more important numeric functions.

ABS

ABS(*n*) returns the absolute value of *n*. For example, the following query returns the absolute value of (price - 20.00) as the third column.

```
SELECT p_code, price, (price - 20), ABS(price - 20.00) FROM product;
```

Applying the query to this product table

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 1      | Nail      | 10.00 | 2013-03-31 |
| 2      | Washer    | 15.00 | 2013-03-29 |
| 3      | Nut       | 15.00 | 2013-03-29 |
| 4      | Screw     | 25.00 | 2013-03-30 |
| 5      | Super_Nut | 30.00 | 2013-03-30 |
| 6      | New Nut   | 30.00 | 2013-05-01 |
+--------+-----------+-------+------------+
```

you will get this result.

```
+--------+-------+--------------+--------------------+
| p_code | price | (price - 20) | ABS(price - 20.00) |
+--------+-------+--------------+--------------------+
| 1      | 10.00 |       -10.00 |              10.00 |
| 2      | 15.00 |        -5.00 |               5.00 |
| 3      | 15.00 |        -5.00 |               5.00 |
| 4      | 25.00 |         5.00 |               5.00 |
| 5      | 30.00 |        10.00 |              10.00 |
| 6      | 30.00 |        10.00 |              10.00 |
+--------+-------+--------------+--------------------+
```

ROUND

ROUND(*n*, *d*) returns a number rounded to a certain number of decimal places. The argument *n* is the number to be rounded and *d* the number of decimal places. For example, the following query uses ROUND to round price to one decimal place.

```
SELECT p_code, price, ROUND (price, 1) FROM product;
```

Assuming the product table contains these rows

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 1      | Nail      | 10.15 | 2013-03-31 |
| 2      | Washer    | 15.99 | 2013-03-29 |
| 3      | Nut       | 15.50 | 2013-03-29 |
| 4      | Screw     | 25.25 | 2013-03-30 |
| 5      | Super_Nut | 30.33 | 2013-03-30 |
| 6      | New Nut   | 15.55 | 2013-05-01 |
+--------+-----------+-------+------------+
```

the output of the query is this.

```
+--------+-------+------------------+
| p_code | price | ROUND (price, 1) |
+--------+-------+------------------+
| 1      | 10.15 |             10.2 |
| 2      | 15.99 |             16.0 |
| 3      | 15.50 |             15.5 |
| 4      | 25.25 |             25.3 |
| 5      | 30.33 |             30.3 |
| 6      | 15.55 |             15.6 |
+--------+-------+------------------+
```

SIGN

SIGN(*n*) returns a value indicating the sign of *n*. This function returns -1 for *n* < 0, 0 for *n* = 0, and 1 for *n* > 0. As an example, the following query uses SIGN to return the sign of (price – 15).

```
SELECT p_code, price, SIGN(price - 15) FROM product;
```

Assuming the **product** table has the following records

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 1      | Nail      | 10.00 | 2013-03-31 |
| 2      | Washer    | 15.00 | 2013-03-29 |
| 3      | Nut       | 15.00 | 2013-03-29 |
| 4      | Screw     | 25.00 | 2013-03-30 |
| 5      | Super_Nut | 30.00 | 2013-03-30 |
| 6      | New Nut   | 15.00 | 2013-05-01 |
+--------+-----------+-------+------------+
```

the query output will be as follows.

```
+--------+-------+------------------+
| p_code | price | SIGN(price - 15) |
+--------+-------+------------------+
| 1      | 10.00 |               -1 |
| 2      | 15.00 |                0 |
| 3      | 15.00 |                0 |
| 4      | 25.00 |                1 |
| 5      | 30.00 |                1 |
| 6      | 15.00 |                0 |
+--------+-------+------------------+
```

TRUNC

TRUNC(*n*, *d*) returns a number truncated to a certain number of decimal places. The argument *n* is the number to truncate and *d* the number of decimal places. For example, the following query truncates price to one decimal place.

```
SELECT p_code, price, TRUNCATE(price, 1) FROM product;
```

Assuming the **product** table contains these rows

```
+--------+----------+-------+------------+
| p_code | p_name   | price | launch_dt  |
+--------+----------+-------+------------+
| 1      | Nail     | 10.15 | 2013-03-31 |
| 2      | Washer   | 15.99 | 2013-03-29 |
| 3      | Nut      | 15.50 | 2013-03-29 |
| 4      | Screw    | 25.25 | 2013-03-30 |
| 5      | Super_Nut | 30.33 | 2013-03-30 |
| 6      | New Nut  | 15.55 | 2013-05-01 |
+--------+----------+-------+------------+
```

the query result will be as follows.

```
+--------+-------+-------------------+
| p_code | price | TRUNCATE(price, 1) |
+--------+-------+-------------------+
| 1      | 10.15 |              10.1 |
| 2      | 15.99 |              15.9 |
| 3      | 15.50 |              15.5 |
| 4      | 25.25 |              25.2 |
| 5      | 30.33 |              30.3 |
| 6      | 15.55 |              15.5 |
+--------+-------+-------------------+
```

Character Functions

The following are some of the more important string functions.

CONCAT

CONCAT(*string1*, *string2*) concatenates *string1* and *string2* and returns the result. If you pass a number as an argument, the number will first be converted to a string. In the following example three strings, *p_name*, a dash, and *description*, are concatenated.

```
SELECT p_code, CONCAT(CONCAT(p_name, ' -- ') , price) FROM product;
```

The **price** column values will be converted automatically to strings.

With the **product** table containing these rows

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 1      | Nail      | 10.15 | 2013-03-31 |
| 2      | Washer    | 15.99 | 2013-03-29 |
| 3      | Nut       | 15.50 | 2013-03-29 |
| 4      | Screw     | 25.25 | 2013-03-30 |
| 5      | Super_Nut | 30.33 | 2013-03-30 |
| 6      | New Nut   | 15.55 | 2013-05-01 |
+--------+-----------+-------+------------+
```

executing the query against the **product** table will return this result.

```
+--------+-------------------------------------+
| p_code | CONCAT(CONCAT(p_name, ' -- ') , price) |
+--------+-------------------------------------+
| 1      | Nail -- 10.15                       |
| 2      | Washer -- 15.99                     |
| 3      | Nut -- 15.50                        |
| 4      | Screw -- 25.25                      |
| 5      | Super_Nut -- 30.33                  |
| 6      | New Nut -- 15.55                    |
+--------+-------------------------------------+
```

You can also use the || operator to concatenate strings. The following query produces the same output as the one above.

```
SELECT p_code, p_name || ' -- ' || price FROM product;
```

LOWER and UPPER

LOWER(*str*) converts *str* to lowercase and UPPER(*str*) converts *str* to uppercase. For example, the following query uses LOWER and UPPER.

```
SELECT p_name, LOWER(p_name), UPPER(p_name) FROM product;
```

Executing the query against the **product** table gives you this result.

```
+-----------+--------------+---------------+
| p_name    | LOWER(p_name) | UPPER(p_name) |
+-----------+--------------+---------------+
| Nail      | nail          | NAIL          |
| Washer    | washer        | WASHER        |
| Nut       | nut           | NUT           |
| Screw     | screw         | SCREW         |
| Super_Nut | super_nut     | SUPER_NUT     |
| New Nut   | new nut       | NEW NUT       |
+-----------+--------------+---------------+
```

LENGTH

LENGTH(*str*) returns the length of string *str*. The length of a string is the number of characters in it. For example, the following query returns the length of **p_name** as the second column.

```
SELECT p_name, LENGTH(p_name) FROM product;
```

The result would look like this.

```
+-----------+----------------+
| p_name    | LENGTH(p_name) |
+-----------+----------------+
| Nail      |              4 |
| Washer    |              6 |
| Nut       |              3 |
| Screw     |              5 |
| Super_Nut |              9 |
| New Nut   |              7 |
+-----------+----------------+
```

SUBSTR

SUBSTR(*str*, *start_position*, [*length*]) returns a substring of *str* starting from the position indicated by *start_position*. If *length* is not specified, the function returns a substring from *start_position* to the last character in *str*. If *length* is present, the function returns a substring which is *length* characters long starting from *start_position*. If *length* is less than 1, the function returns an empty string.

Suppose you have a **customer** table with the following rows.

```
+------+----------------+----------------+
| c_no | c_name         | phone          |
+------+----------------+----------------+
| 10   | Standard Store | 1-416-223-4455 |
| 20   | Quality Store  | 1-647-333-5566 |
| 30   | Head Office    | 1-416-111-2222 |
| 40   | Super Agent    | 1-226-777-8888 |
+------+----------------+----------------+
```

The following query

```
SELECT SUBSTR(phone, 3) FROM customer;
```

will return the following result

```
+------------------+
| SUBSTR(phone, 3) |
+------------------+
| 416-223-4455     |
| 647-333-5566     |
| 416-111-2222     |
| 226-777-8888     |
+------------------+
```

And the following query

```
SELECT CONCAT( SUBSTR(phone, 7, 3) , SUBSTR(phone, 11, 3)) phone
FROM customer;
```

will return this result.

```
+--------+
| phone  |
+--------+
| 223445 |
| 333556 |
| 111222 |
| 777888 |
+--------+
```

Datetime Functions

The following are some of the more important datetime functions.

CURRENT_DATE

CURRENT_DATE() returns the current date (the current date of the MySQL server at the time you run the query). For instance, the following query

```
SELECT p_code, launch_dt, CURRENT_DATE FROM product;
```

will return a result that looks like this. The actual value of the third column will depend on when you run the query.

```
+--------+------------+--------------+
| p_code | launch_dt  | CURRENT_DATE |
+--------+------------+--------------+
| 1      | 2013-03-31 | 2013-09-16   |
| 2      | 2013-03-29 | 2013-09-16   |
| 3      | 2013-03-29 | 2013-09-16   |
| 4      | 2013-03-30 | 2013-09-16   |
| 5      | 2013-03-30 | 2013-09-16   |
| 6      | 2013-05-01 | 2013-09-16   |
+--------+------------+--------------+
```

DATE_FORMAT

TO_CHAR(*dt*, *fmt_specifier*) converts a date (*dt*) to a string in the format specified by *fmt_specifier*. In the following example, the **launch_dt** column is formatted with a format specifier that has three components:

- %d - the day of the month
- %M - the long name of the month
- %Y - the year

```
SELECT p_code, DATE_FORMAT(launch_dt, '%d %M %Y') reformatted_dt
FROM product;
```

Running the query will give you something like this.

```
+--------+----------------+
| p_code | reformatted_dt |
+--------+----------------+
| 1      | 31 March 2013  |
| 2      | 29 March 2013  |
| 3      | 29 March 2013  |
| 4      | 30 March 2013  |
| 5      | 30 March 2013  |
| 6      | 01 May 2013    |
+--------+----------------+
```

NULL-related functions

The following are some of the functions that can be used to handle null values.

COALESCE

COALESCE(*expr-1*, *expr-2*, ..., *expr-n*) returns the first expression from the list that is not NULL. For example, suppose your **product** table contains the following rows

```
+--------+-----------+-------+------------+-----------+
| p_code | p_name    | price | launch_dt  | min_price |
+--------+-----------+-------+------------+-----------+
| 1      | Nail      | 10.15 | 2013-03-31 |      NULL |
| 2      | Washer    | 15.99 | 2013-03-29 |      NULL |
| 3      | Nut       | 15.50 | 2013-03-29 |     12.00 |
| 4      | Screw     | 25.25 | 2013-03-30 |     17.00 |
| 5      | Super_Nut |  NULL | 2013-03-30 |     10.00 |
| 6      | New Nut   |  NULL | 2013-05-01 |      NULL |
+--------+-----------+-------+------------+-----------+
```

and you want to view the **sale_price** column of the products using this formula:

- If price is available (not NULL) then discount it by 10%
- If price is not available then return min_price
- If both price and min_price are not available, return 5.0

You can use COALESCE to produce the correct **sale_price** values:

```
SELECT p_name, price, min_price,
COALESCE((price * 0.9), min_price, 5.0) sale_price
FROM product;
```

Here is the query result.

```
+-----------+-------+-----------+------------+
| p_name    | price | min_price | sale_price |
+-----------+-------+-----------+------------+
| Nail      | 10.15 |      NULL |      9.135 |
| Washer    | 15.99 |      NULL |     14.391 |
| Nut       | 15.50 |     12.00 |     13.950 |
| Screw     | 25.25 |     17.00 |     22.725 |
| Super_Nut |  NULL |     10.00 |     10.000 |
| New Nut   |  NULL |      NULL |      5.000 |
+-----------+-------+-----------+------------+
```

NULLIF

NULLIF (*expr1*, *expr2*) compares *expr1* and *expr2*. If they are equal, the function returns null. If they are not equal, the function returns *expr1*.

Suppose you stored old product prices in a table named **old_price**. The following **old_price** table, for example, shows two old prices of the Nut product, the products with p_code = 3.

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 3      | Nut       | 15.00 | 2013-03-01 |
| 3      | Newer Nut | 12.00 | 2013-04-01 |
+--------+-----------+-------+------------+
```

Just say you want to show the old products with their current price. The following query that employs the NULLIF function can solve your problem.

```
SELECT p.p_code,
  p.p_name,
  NULLIF(p.price, op.price) current_price
FROM product p
JOIN old_price op USING (p_code);
```

Applying the query against the following product table and the **old_price** table

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 1      | Nail      | 10.15 | 2013-03-31 |
| 2      | Washer    | 15.99 | 2013-03-29 |
| 3      | Nut       | 15.50 | 2013-03-29 |
| 4      | Screw     | 25.25 | 2013-03-30 |
| 5      | Super_Nut | 30.33 | 2013-03-30 |
| 6      | New Nut   | 15.55 | 2013-05-01 |
+--------+-----------+-------+------------+
```

returns the two old nuts as follows

```
+--------+--------+---------------+
| p_code | p_name | current_price |
+--------+--------+---------------+
| 3      | Nut    |         15.50 |
| 3      | Nut    |         15.50 |
+--------+--------+---------------+
```

IFNULL

IFNULL(*expr1*, *expr2*) returns *exprs1* if *expr1* is not NULL; otherwise, it returns *expr2*.

For example, suppose you want to compare old and current prices. Applying NVL in the following query gives you the old price from the product table if the product has never been superseded; otherwise, if a product has been superseded, its old price will come from the old_product table.

```
SELECT p_code,
  p.p_name,
  p.price current_price,
  IFNULL(op.price,p.price) old_price
FROM product p
LEFT OUTER JOIN old_product op USING (p_code);
```

Here is the result.

```
+--------+-----------+---------------+-----------+
| p_code | p_name    | current_price | old_price |
+--------+-----------+---------------+-----------+
| 3      | Nut       |         15.50 |     15.00 |
| 3      | Nut       |         15.50 |     12.00 |
| 1      | Nail      |         10.15 |     10.15 |
| 2      | Washer    |         15.99 |     15.99 |
| 4      | Screw     |         25.25 |     25.25 |
| 5      | Super_Nut |          NULL |      NULL |
| 6      | New Nut   |          NULL |      NULL |
+--------+-----------+---------------+-----------+
```

The query result shows that the only product that has been superseded is Nut, and it has been superseded twice. Therefore, its old prices are shown from the **old_product** table. The other products have never been superseded so their current and old prices are the same. Their **old_price** is coming from the product table because its **op.price** column, the first parameter of the NVL function, is NULL.

Summary

You learned some of the built-in functions that you can use in the MySQL database. If you are interested in learning more about built-in functions, consult the MySQL manual, available at MySQL website.

Chapter 11 Transactions

If you have a sequence of SQL statements that must either complete successfully or have no effect whatsoever, you need a transaction. You will learn about transactions in this chapter.

Overview

A series of SQL statements that must be treated as a unit are called a transaction or work. The deal with a transaction is it has a all-or-nothing proposition. Either all the statements complete successfully or none of them has effect at all.

For example, when making an online money transfer from one bank account to another, you need to ensure that both of these operations are successful or neither is committed.

- Money is taken from the sender's account.
- The recipient's account is credited by the same amount.

Both statements must be completed successfully. If the second statement fails, any changes done by the first statement must be rolled back.

As another example of statements that require a transaction, consider an order shipment. When shipping a product in an order, you must

- Deduct the product quantity from the inventory, and
- Change the order status to "Shipped".

To make multiple SQL statements act as a single transaction, you need to issue a START TRANSACTION statement before the first update statement.

```
START TRANSACTION;
```

You can now issue the first update statement like this (reducing the quantity by ship_qty).

```
UPDATE p_inventory SET p_qty = (p_qty - ship_qty)
WHERE c_no = 10 AND p_code = 1;
```

If, for example, this causes p_qty to have a negative value (which is not allowed), this statement must be rolled back. In MySQL, you can roll back a transaction using the ROLLBACK statement.

```
ROLLBACK;
```

Once you issue a ROLLBACK statement, the transaction unit is no longer in effect; you must issue a START TRANSACTION again, which starts a new transaction unit.

If the update was successful, you can issue the second statement to update the product inventory table:

```
UPDATE c_order SET ship_flag = 'Y' WHERE c_no = 10 and p_code = 1;
```

If this second statement failed, issue a ROLLBACK statement similar and both updates will get rolled back.

If both updates were successful, issue a COMMIT statement as follows.

```
COMMIT;
```

This COMMIT statement persists the data changes. Like the ROLLBACK statement, once you issue a COMMIT statement, the transaction unit is no longer in effect.

SAVEPOINT

You can issue a SAVEPOINT statement in between multiple SQL statements. For example, after two successful updates, you might want to insert a record for shipment reporting. If the insert fails and you want to roll back only the insert, you can apply a savepoint in the transaction unit.

The syntax of for the SAVEPOINT statement is as follows.

```
SAVEPOINT savepoint_name;
```

The following is an example of transaction with a savepoint.

```
START TRANSACTION;

UPDATE p_inventory SET p_qty = (p_qty - ship_qty) WHERE c_no = 10
      AND p_code = 1;

UPDATE c_order SET ship_flag = 'Y' WHERE c_no = 10 AND p_code = 1;

SAVEPOINT shipped;

INSERT INTO shipment_report (10, 1, ship_qty, month_year)
VALUES(10, 1, 100, '05-2013');

ROLLBACK TO SAVEPOINT shipped;
```

You issue the ROLLBACK TO SAVE POINT statement if your last insert failed. It will cancel the last insert statement only. The two updates are still in effect, waiting for you to commit or roll them back.

If the last insert statement is successful, you can issue a COMMIT statement, followed by a

```
RELEASE SAVEPOINT shipped;
```

A RELEASE SAVEPOINT statement removes the savepoint, to make sure the savepoint is no longer in effect.

AutoCommit

Issuing a START TRANSACTION statement sets the autocommit mode off (autocommit = 0) for that transaction. When autocommit is off, the changes done by insert/update/delete statements in the transaction will not be persisted until you issue a COMMIT statement. Once the COMMIT statement is executed, the autocommit set by the START TRANSACTION is no longer effective. A ROLLBACK statement also terminates an autocommit mode.

If you need to, you can set the autocommit mode across transaction units by issuing a SET AUTOCOMMIT command. The syntax for the SET AUTOCOMMIT command is as follows.

```
SET AUTOCOMMIT = mode;
```

where *mode* is either 0 (off) or 1 (on).

Summary

In this chapter you learned how to use the SQL transactional statements to accomplish safe data changes. In the next chapter you will learn about stored routines, which extend SQL with procedural language features. You will see how a transaction is handled within a stored routine.

Chapter 12 Stored Routines

A stored routine, which can be a function or procedure, is a set of SQL statements stored in the MySQL database server. Stored routines have procedural programming features complementing the data manipulation SQL statements. Stored routines are a feature of the MySQL database.

The objective of this chapter is to introduce some of the most commonly used stored routine features such as

- row-by-row processing of query output
- if-then-else decision logic
- exception handling
- user-defined functions

Row-by-row Processing

You learned in the previous chapters that the output of a query can be more than one row. You can write a procedure to process query output row-by-row sequentially.

The structure of a procedure for row-by-row processing is as follows.

```
delimiter characters
CREATE PROCEDURE name(parameters)
BEGIN
DECLARE variables;
DECLARE cursor;
DECLARE exception_handler;
OPEN cursor;
loop_label LOOP
FETCH cursor INTO variables
  exception_handling statement;
  processing_statements;
  END LOOP;
CLOSE cursor;
END;
```

For example, the program in Listing 12.1 creates a procedure named write_invoice. The write_invoice procedure makes use of a join query declared as a cursor named invoice_csr. The cursor holds the rows (invoice rows) returned by the query. Each of these rows is then inserted into the **invoice** table.

Listing 12.1: Row-by-row processing

```
delimiter $$
CREATE PROCEDURE write_invoice()
BEGIN
DECLARE no_more_row INTEGER;
DECLARE c_no_var, p_no_var VARCHAR(6);
DECLARE c_name_var VARCHAR(40);
DECLARE p_name_var VARCHAR(40);
DECLARE qty_var INTEGER;
DECLARE price_var DECIMAL(4,2);
DECLARE total_price DECIMAL(10,2);
DECLARE invoice_dt DATE;

DECLARE invoice_csr CURSOR FOR
     SELECT c_no,
     c_name,
     p_name,
     qty,
     price  ,
     (qty * price)  ,
     DATE_FORMAT(SYSDATE(), '%Y-%m-%d')
  FROM c_order NATURAL
  JOIN product NATURAL
  JOIN customer
;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET no_more_row=1;

OPEN invoice_csr;
csr_loop: LOOP
        FETCH invoice_csr INTO c_no_var, c_name_var
,p_name_var,qty_var,price_var,total_price,invoice_dt;
IF no_more_row = 1 THEN
        LEAVE csr_loop;
```

```
END IF;
    INSERT INTO invoice VALUES(c_no_var,c_name_var, p_name_var
, qty_var,price_var,total_price,invoice_dt);
END LOOP;
    CLOSE invoice_csr;
END $$
```

Suppose you have **c_order**, **customer**, and **product** tables like the ones shown in Tables 12.1, 12.2, and 12.3, respectively, and suppose you also have an empty **invoice** table.

```
+------+--------+------+------------+
| C_NO | P_CODE | QTY  | ORDER_DT   |
+------+--------+------+------------+
| 10   | 1      |  100 | 2013-04-01 |
| 10   | 2      |  100 | 2013-04-01 |
| 20   | 1      |  200 | 2013-04-01 |
| 40   | 4      |  400 | 2013-04-02 |
| 40   | 5      |  400 | 2013-04-03 |
+------+--------+------+------------+
```

Table 12.1: The c_order table

```
+------+----------------+
| c_no | c_name         |
+------+----------------+
| 10   | Standard Store |
| 20   | Quality Store  |
| 30   | Head Office    |
| 40   | Super Agent    |
+------+----------------+
```

Table 12.2: The customer table

```
+--------+-----------+-------+------------+
| p_code | p_name    | price | launch_dt  |
+--------+-----------+-------+------------+
| 1      | Nail      | 10.00 | 2013-03-31 |
| 2      | Washer    | 15.00 | 2013-03-29 |
| 3      | Nut       | 15.00 | 2013-03-29 |
| 4      | Screw     | 25.00 | 2013-03-30 |
| 5      | Super_Nut | 30.00 | 2013-03-30 |
| 6      | New Nut   |  NULL | NULL       |
+--------+-----------+-------+------------+
```

Table 12.3: The product table

When you execute the query in Listing 12.1, the write_invoice procedure will be created and stored in the database. You can then call the procedure as follows.

```
CALL write_invoice();
```

The **invoice** table will be populated with the following rows.

```
+------+----------------+-----------+------+-------+-------------+
| c_no | c_name         | p_name    | qty  | price | total_price |
+------+----------------+-----------+------+-------+-------------+
| 10   | Standard Store | Nail      |  100 | 10.00 |     1000.00 |
| 10   | Standard Store | Washer    |  100 | 15.00 |     1500.00 |
| 20   | Quality Store  | Nail      |  200 | 10.00 |     2000.00 |
| 30   | Head Office    | Nut       |  300 | 15.00 |     4500.00 |
| 40   | Super Agent    | Screw     |  400 | 25.00 |    10000.00 |
| 40   | Super Agent    | Super_Nut |  400 | 30.00 |    12000.00 |
+------+----------------+-----------+------+-------+-------------+
```

If-Then-Else Decision Logic

You can use an if statement to branch in a program. For instance, in Listing 12.2 I have added an if-then-else decision logic to the invoice row insertion of Listing 12.1. In Listing 12.2 the total price is discounted if the it is => 10000.

Listing 12.2: If-then-else decision logic

```
delimiter $$
CREATE PROCEDURE write_invoice_with_ifthenelse()
BEGIN
DECLARE no_more_row INTEGER;
DECLARE c_no_var, p_no_var VARCHAR(6);
DECLARE c_name_var VARCHAR(40);
DECLARE p_name_var VARCHAR(40);
DECLARE qty_var INTEGER;
DECLARE price_var DECIMAL(4,2);
DECLARE total_price DECIMAL(7,2);

DECLARE invoice_csr CURSOR FOR
     SELECT c_no,
     c_name,
     p_name,
     qty,
     price  ,
     (qty * price)
  FROM c_order NATURAL
  JOIN product NATURAL
  JOIN customer
;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET no_more_row=1;

OPEN invoice_csr;
csr_loop: LOOP
        FETCH invoice_csr INTO c_no_var, c_name_var
,p_name_var,qty_var,price_var,total_price;
IF no_more_row = 1 THEN
        LEAVE csr_loop;
END IF;
IF total_price < 10000 THEN
    INSERT INTO invoice VALUES(c_no_var,c_name_var, p_name_var
, qty_var,price_var,total_price);
 ELSE
   INSERT INTO invoice VALUES(c_no_var,c_name_var, p_name_var
 , qty_var,price_var, total_price - (total_price * 0.01));
END IF;
END LOOP;
    CLOSE invoice_csr;
END $$
```

For this example, I'm using a **c_order** table with the following rows. The difference between this **c_order** table and the one used in the previous example is this table has a sixth row that records the sale of a product (p_co = 6) with a null **launch_dt**.

```
+------+--------+------+------------+
| C_NO | P_CODE | QTY  | ORDER_DT   |
+------+--------+------+------------+
| 10   | 1      |  100 | 2013-04-01 |
| 10   | 2      |  100 | 2013-04-01 |
| 20   | 1      |  200 | 2013-04-01 |
| 40   | 4      |  400 | 2013-04-02 |
| 40   | 5      |  400 | 2013-04-03 |
| 40   | 6      |  600 | 2013-05-01 |
+------+--------+------+------------+
```

When you call the procedure, the following rows will now be inserted into the **invoice** table.

```
+------+----------------+-----------+------+-------+-------------+
| c_no | c_name         | p_name    | qty  | price | total_price |
+------+----------------+-----------+------+-------+-------------+
| 10   | Standard Store | Nail      |  100 | 10.00 |     1000.00 |
| 10   | Standard Store | Washer    |  100 | 15.00 |     1500.00 |
| 20   | Quality Store  | Nail      |  200 | 10.00 |     2000.00 |
| 30   | Head Office    | Nut       |  300 | 15.00 |     4500.00 |
| 40   | Super Agent    | Screw     |  400 | 25.00 |     9900.00 |
| 40   | Super Agent    | Super_Nut |  400 | 30.00 |    11880.00 |
+------+----------------+-----------+------+-------+-------------+
```

Exception Handling

Stored routines allow you to handle errors (or exceptions) in the program. Listings 12.1 and 12.2 have exception-handlers. You declare an exception handler as follows. Notice the no_more_row variable, which will be set to 1 when the exception occurs.

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET no_more_row = 1;
```

Then inside the processing logic where you anticipate the exception might occur, you check the value of the handler variable. In our examples, if the value of the variable is 1, the program exits (leave) the loop.

User-defined Functions

You learned MySQL built-in functions in Chapter 10, “Built-in Functions.” You can write your own function as a stored routine. The syntax for creating a user-defined function is as follows.

```
CREATE FUNCTION FUNCTION name(parameters)
    RETURNS data_type
  BEGIN
    processing_statements, including a RETURN statement
  END;
```

For example, Listing 12.3 shows a user-defined function named calc_new_price. The function takes two parameters and uses the values of the parameters to calculate a new price and returns the result. If you execute the statement in Listing 12.3, a stored function named calc_new_price will be created and stored in your database.

Listing 12.3: Creating function calc_new_price

```
delimiter $$
CREATE FUNCTION calc_new_price(
      exist_price DECIMAL,
      change_percentage DECIMAL(10,4))
   RETURNS DECIMAL(10,4)
   BEGIN
    RETURN exist_price + (exist_price * change_percentage);
  END $$
```

Now you can use the function just like you would any MySQL built-in function. The update statement in Listing 10.4, for example, uses the calc_new_price function to calculate new prices and update the product prices.

Listing 12.4: Using the calc_new_price function

```
UPDATE product SET price = calc_new_price(price, 0.1) ;
```

Transactions

You learned about transactions in Chapter 11, “Transactions”. You can define a transaction unit in a stored routine.

Listing 12.5 presents an example of a stored routine that contains a transaction. The transaction unit starts with a START TRANSACTION statement and ends with the END IF statement. It consists of two UPDATE statements. If the part inventory falls below zero the update to the inventory is rolled back; otherwise the update to the **c_order** and **inventory** tables are committed.

Listing 12.5: Transaction Unit

```
delimiter $$
CREATE PROCEDURE transaction_unit(ship_qty INTEGER)
BEGIN
DECLARE p_qty_var INTEGER;
START TRANSACTION;
SELECT p_qty INTO p_qty_var FROM p_inventory WHERE p_code = 1;
```

```
UPDATE p_inventory SET p_qty = (p_qty - ship_qty) WHERE p_code = 1;
IF (p_qty_var - ship_qty) < 0 THEN
 ROLLBACK;
 ELSE
UPDATE c_order SET ship_flag = 'Y' WHERE c_no = 10 AND p_code = 1;
COMMIT;
 END IF;
END;
```

Summary

In this chapter you learned about several stored routine features. However, what's presented here is just the tip of the iceberg. There are so many features that you can use in stored routines to help you with your real-world application development. These other features are unfortunately beyond the scope of this book.

Chapter 13
The Data Dictionary

The data dictionary of a database system contains data about the data in the database. As such, a data dictionary is also known as metadata. MySQL metadata is stored in a database named *information_schema*. Just like your data, the metadata is stored as tables in the information_schema database. Therefore, you can use your SQL skills gained so far to query the tables in it. The metadata is maintained by the MySQL database system. You can only read the metadata; you cannot add data or change any of the metadata.

In this chapter you explore some of the tables in the information_schema database.

The Schemata Table

The *schemata* table is one of the tables in the information_schema database. This table stores the information about all the databases on the MySQL server.

The query in Listing 13.1 can be used to list all the databases on your MySQL server, including the information_schema database itself. The query selects only the schema_name column. To see the columns of a table, issue a *SHOW COLUMNS FROM table_name* command. You can then select which ones to include in the query. Note that in MySQL the term schema is used to refer to a database.

Listing 13.1: Querying the Schemata

```
SELECT schema_name FROM information_schema.schemata;
```

Here is the output of the query in Listing 11.1. Remember, what you see on your computer may be different. The output is the list of databases you have on your MySQL server when you query the schemata table.

```
+--------------------+
| schema_name        |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sales              |
+--------------------+
```

The Tables Table

The *tables* table stores the names and other information of your tables. Use the query in Listing 11.2 to see the tables and views that you have in the *sales* database. The query selects the names and the table types and includes views.

Listing 13.2: Querying the Tables

```
SELECT
    table_name, table_type
FROM
    information_schema.tables
WHERE
    table_schema = 'sales';
```

Again, the query output from running the query in Listing 11.2 in your MySQL server may not be exactly the same as what I have got here. Here is mine.

```
+-----------------+------------+
| table_name      | table_type |
+-----------------+------------+
| c_order         | BASE TABLE |
| customer        | BASE TABLE |
| invoice         | BASE TABLE |
| non_nut         | BASE TABLE |
| nut_product     | BASE TABLE |
| old_price       | BASE TABLE |
| prod_subs_v     | VIEW       |
| prod_subst      | BASE TABLE |
| product         | BASE TABLE |
| product_sell_v  | VIEW       |
| product_v       | VIEW       |
| shipment        | BASE TABLE |
+-----------------+------------+
```

The Columns Table

You can get the names of columns from the *columns* table. For example, the query in Listing 11.3 shows you the column names of the **product** table and their data type, maximum length, and the physical position (called ordinal position) of every column in the table.

Listing 13.3: Querying the Columns table

```
SELECT column_name, column_type, character_maximum_length max_len,
ordinal_position ord_pos
FROM information_schema.columns
WHERE table_schema = 'sales' and table_name = 'product';
```

Here is the query output.

```
+-------------+--------------+---------+---------+
| column_name | column_type  | max_len | ord_pos |
+-------------+--------------+---------+---------+
| p_code      | varchar(6)   |       6 |       1 |
| p_name      | varchar(15)  |      15 |       2 |
| price       | decimal(4,2) |    NULL |       3 |
| launch_dt   | date         |    NULL |       4 |
| margin      | decimal(6,2) |    NULL |       5 |
| s_code      | varchar(45)  |      45 |       6 |
+-------------+--------------+---------+---------+
```

The Routines Table

This table contains information about the routines (stored functions and procedures). For example, the query in Listing 11.6 returns the routines in the sales database.

Listing 13.4: Querying the Routines table

```
SELECT routine_name, routine_type
FROM information_schema.routines
WHERE routine_schema = 'sales';
```

Here is the output.

```
+-------------------------------+--------------+
| routine_name                  | routine_type |
+-------------------------------+--------------+
| calc_new_price                | FUNCTION     |
| write_invoice                 | PROCEDURE    |
| write_invoice_with_ifthenelse | PROCEDURE    |
+-------------------------------+--------------+
```

If you need to see the content of a routine, select the routine_definition column of the routine. Executing the query in Listing 13.5 for example will show the code lines of the write_invoice procedure we created in Chapter 12, “Stored Routines”.

Listing 13.5: Querying Code Lines

```
SELECT routine_definition
 FROM information_schema.routines
WHERE routine_schema = 'sales' AND routine_name = 'write_invoice';
```

Summary

The data dictionary contains the metadata of your database. In this chapter you learned to use some of the information schema.

Appendix A
Installing MySQL Community Edition

To try out the examples in this book, you need a MySQL database. Because you need to create tables and other objects, as well as store and update data, it is best if you have your own database. Fortunately, you can download MySQL Community Edition for free from MySQL's website. As you will learn in this appendix, MySQL Community Edition comes with a command line client called *mysql* that you can use to run SQL statements. MySQL is supported on a number of platforms, the following installation guide is for Windows only.

Downloading MySQL Community Edition

This database software can be downloaded from this web page.

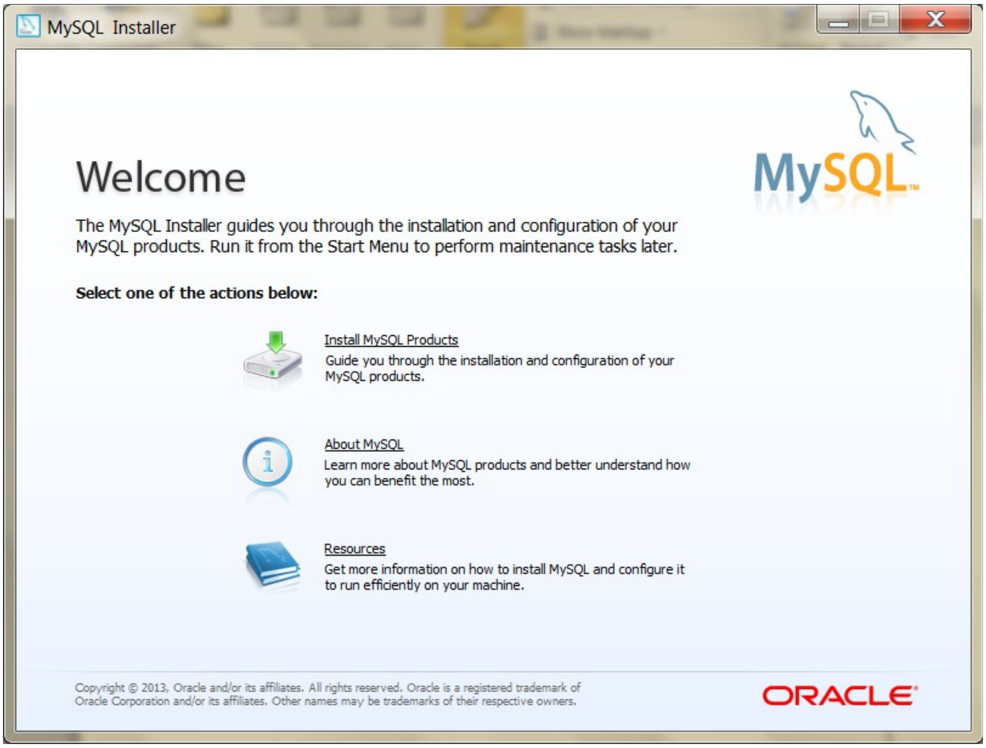
http://dev.mysql.com/downloads/mysql/

Scroll down until you see **MySQL Installer for Windows** on the list. Then, select MySQL Installer MSI by clicking its Download button. On the next web page, click the "No thanks, just start my download" link, at the bottom left corner to download the installer.

Note
The book examples are tested on Windows. They should work equally well on other platforms.

Installing MySQL Community Edition

Double-click on the MSI file you downloaded to run the installer. You will see the MySQL Installer Welcome window like that in Figure A.1.


MySQL Installer

Welcome

The MySQL Installer guides you through the installation and configuration of your MySQL products. Run it from the Start Menu to perform maintenance tasks later.

Select one of the actions below:

Install MySQL Products
Guide you through the installation and configuration of your MySQL products.

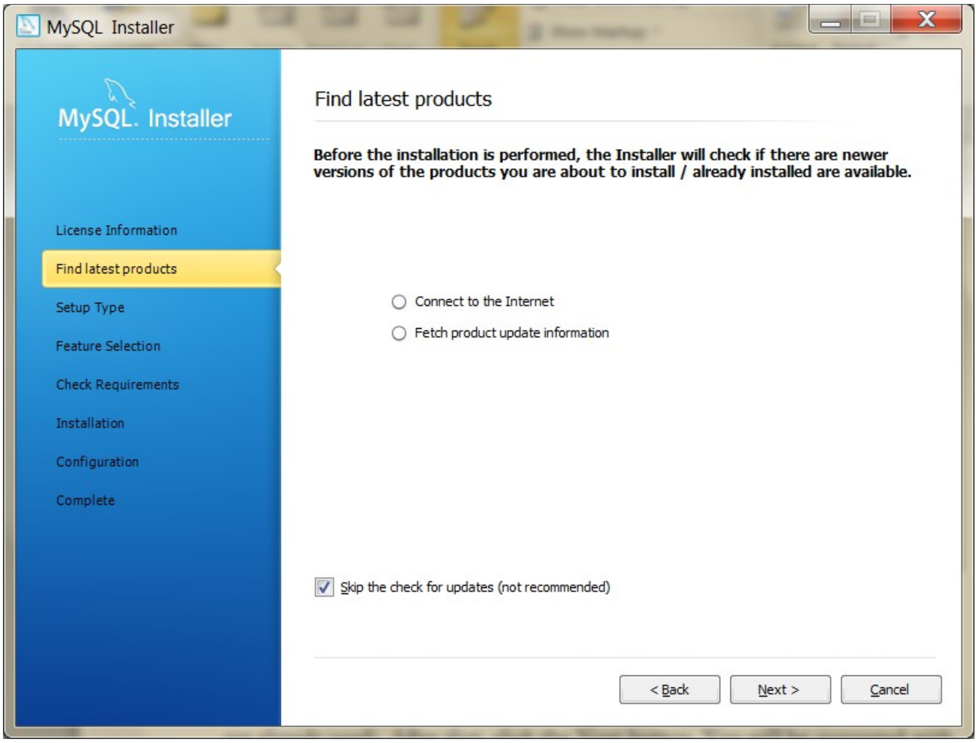
About MySQL
Learn more about MySQL products and better understand how you can benefit the most.

Resources
Get more information on how to install MySQL and configure it to run efficiently on your machine.

Copyright © 2013, Oracle and/or its affiliates. All rights reserved. Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

Figure A.1: The Welcome page of the MySQL Installer

Click the **Install MySQL Products** link, accept the agreement on the License Agreement window, and click the Next button. The next window that will appear is the “Find latest products” window like the one in Figure A.2.



Find latest products

Before the installation is performed, the Installer will check if there are newer versions of the products you are about to install / already installed are available.

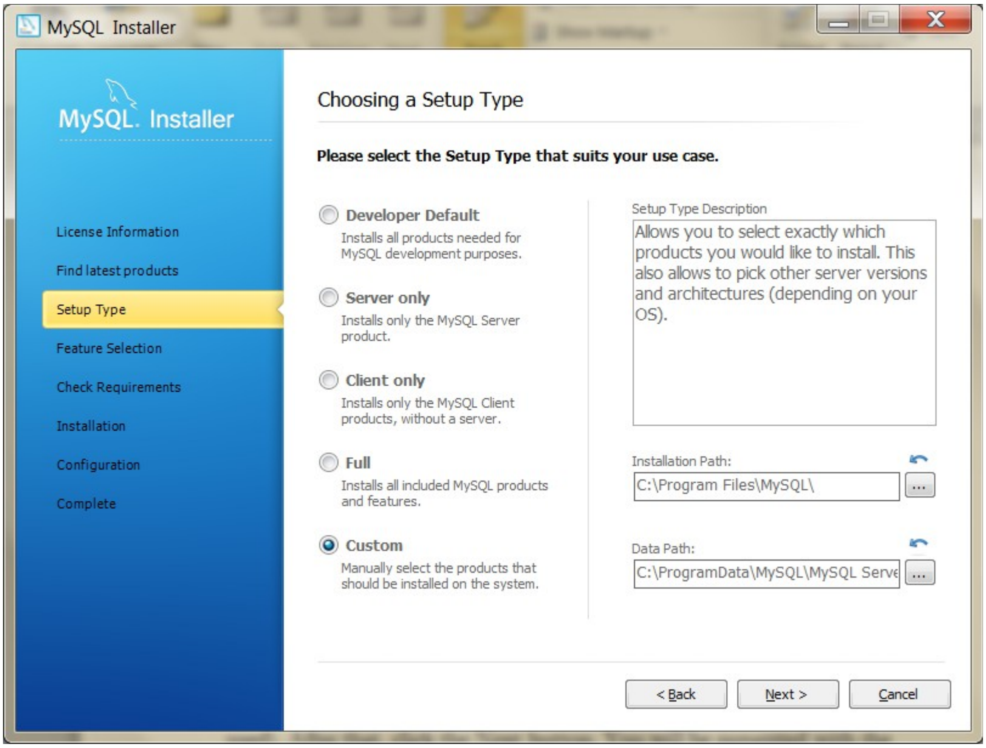
○ Connect to the Internet

○ Fetch product update information

☑ Skip the check for updates (not recommended)

Figure A.2: Latest Products window

Make sure the "Skip the check for updates (not recommended)" radio button is checked and click the Next button. The Setup Type window like that in Figure A.3 will show up.



Choosing a Setup Type

Please select the Setup Type that suits your use case.

( ) Developer Default
Installs all products needed for MySQL development purposes.

( ) Server only
Installs only the MySQL Server product.

( ) Client only
Installs only the MySQL Client products, without a server.

( ) Full
Installs all included MySQL products and features.

[✓ Custom]
Manually select the products that should be installed on the system.

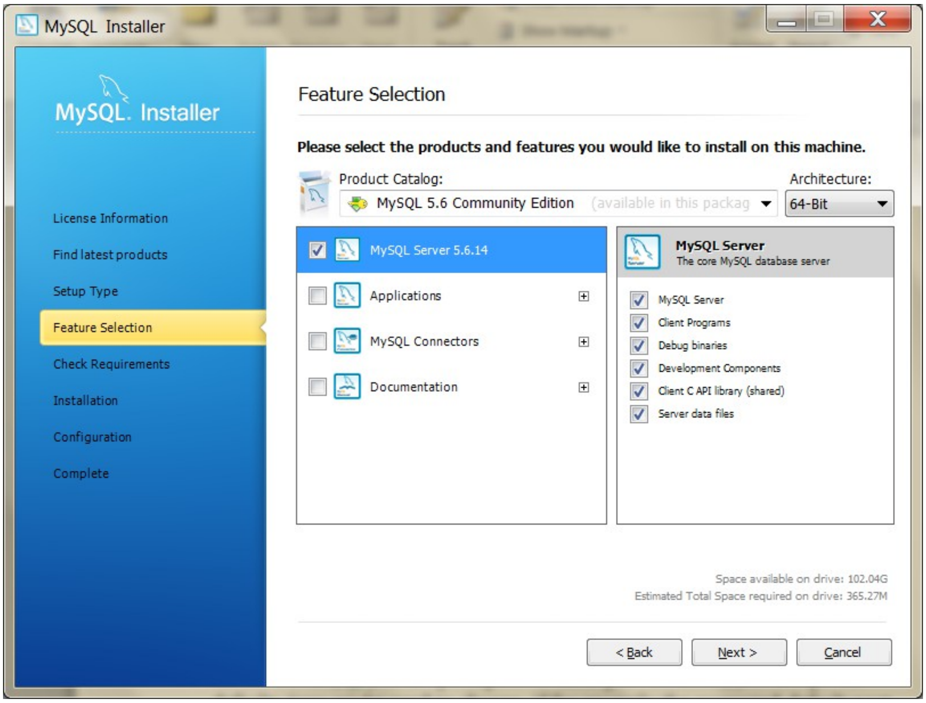
Setup Type Description
Allows you to select exactly which products you would like to install. This also allows to pick other server versions and architectures (depending on your OS).

Installation Path:
C:\Program Files\MySQL\

Data Path:
C:\ProgramData\MySQL\MySQL Serve

Figure A.3: Setup Type window

Select Custom, and you will see the Feature Selection window like that in Figure A4.



Feature Selection

Please select the products and features you would like to install on this machine.

Product Catalog: MySQL 5.6 Community Edition (available in this packag

Architecture: 64-Bit

- [✓] MySQL Server 5.6.14
- [ ] Applications
- [ ] MySQL Connectors
- [ ] Documentation

MySQL Server
The core MySQL database server

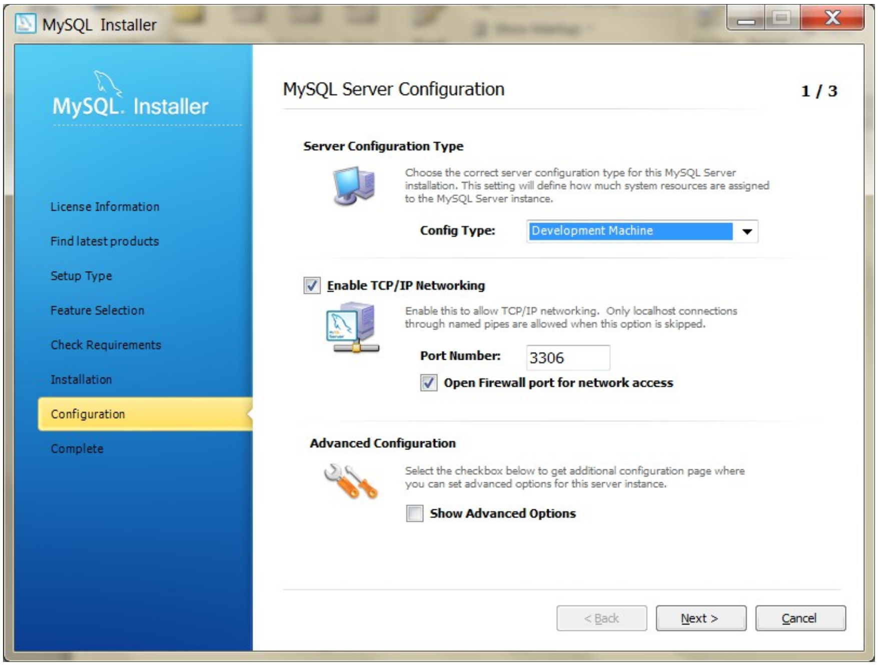
- [✓] MySQL Server
- [✓] Client Programs
- [✓] Debug binaries
- [✓] Development Components
- [✓] Client C API library (shared)
- [✓] Server data files

Space available on drive: 102.04G
Estimated Total Space required on drive: 365.27M

Figure A.4: Feature Selection window

Choose the MySQL Server and Documentation only (the first and last items on the left list) and click the Next button. On the next window that appears, click the Next button again and then click Execute on the next window. On the next two windows, click the Next buttons.

The Configuration window will appear like that in Figure A.5. Click the Next button.



MySQL Server Configuration 1 / 3

Server Configuration Type

Choose the correct server configuration type for this MySQL Server installation. This setting will define how much system resources are assigned to the MySQL Server instance.

Config Type: Development Machine

[✓] Enable TCP/IP Networking

Enable this to allow TCP/IP networking. Only localhost connections through named pipes are allowed when this option is skipped.

Port Number: 3306

[✓] Open Firewall port for network access

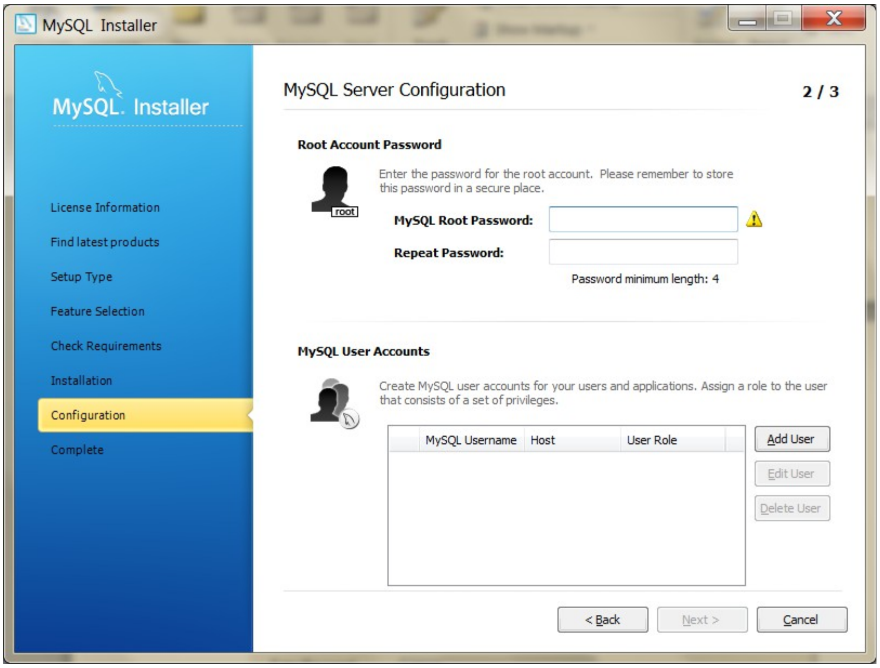
Advanced Configuration

Select the checkbox below to get additional configuration page where you can set advanced options for this server instance.

[ ] Show Advanced Options

Figure A.5: Configuration window

You will be prompted to enter and confirm the root's password (See Figure A.6), then you can click the Next button.


MySQL Installer

MySQL Server Configuration 2 / 3

Root Account Password

Enter the password for the root account. Please remember to store this password in a secure place.

MySQL Root Password:

Repeat Password:

Password minimum length: 4

MySQL User Accounts

Create MySQL user accounts for your users and applications. Assign a role to the user that consists of a set of privileges.

MySQL Username	Host	User Role

Figure A.6: Password window

On the next Configuration window, click the Next button. When the installation finishes, click the Finish button.

An link to the MySQL Command Line Client program will also be added to your Windows Start (See Figure A7).

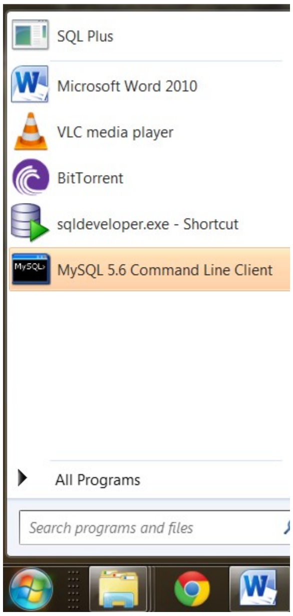
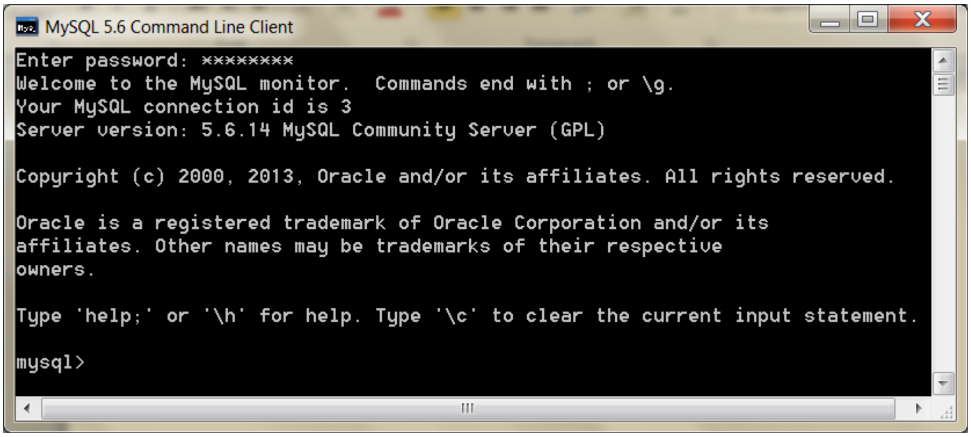

SQL Plus
Microsoft Word 2010
VLC media player
BitTorrent
sqldeveloper.exe - Shortcut
MySQL 5.6 Command Line Client
All Programs
Search programs and files

Figure A.7: The MySQL Command Line Client on the Start program

When you execute (select) the Command Line Client and enter the root password you entered during installation, your client window will look like that in Figure A.8. The client is now ready for you to enter SQL statements and test the book examples. To close the client, enter *exit* on the prompt.



```
Enter password: ********
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3
Server version: 5.6.14 MySQL Community Server (GPL)

Copyright (c) 2000, 2013, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

Figure A.8: Command Line Client is ready

Appendix B
MySQL Built-in Data Types

The MySQL database provides built-in numeric, string, and date and time data types.

Table B.1 summarizes the built-in numeric data types.

Data Type	Description
BIT(*m*)	Bit-field data type for storing binary value of *m* long to a maximum of 64.
TINYINT	Stores integer ranging from -128 to 127
SMALLINT	Stores integer ranging from -32678 to 32677
MEDIUMINT	Stores integer ranging from -8388608 to 8388607
INTEGER	Stores integer ranging from -2.147,483,648 to 2,147,483,647
BIGINT	Stores integer ranging from -9223372036854775808 to 9223372036854775807
DECIMAL(*p*, *s*)	Number having precision *p* and scale *s*
FLOAT	4 bytes floating point number
DOUBLE	8 bytes floating point number
BOOL	A value of zero represents false and nonzero true

Table B.1: MySQL Built-in Numeric Data Types

Table B.2 contains the built-in string types.

Data Type	Description
CHAR(*ml*)	Fixed-length character string having a length of *ml*
VARCHAR(*ml*)	Variable-length character string having a maximum length of *ml*
TEXT	String having a maximum of 65535 characters
TINYTEXT	A text having a maximum of 255 characters
MEDIUMTEXT	A text having a maximum of 16777215 characters
LONGTEXT	A text having a maximum of 4G characters
BLOB/TINYBLOB/MEDIUMBLOB/LONGBLOB	The binary large objects equivalent of the four text data types. Data is treated as binary strings.
ENUM	A string with a value chosen from a list of permitted values
SET	Similar to ENUM. While a column with ENUM data type can store one string, SET can store more than one from the permitted values.

Table B.2: MySQL Built-in Data Types

Finally, Table B.3 contains the date and time built-in data types.

Data Type	Description
DATE	Valid date ranging from 1000-01-01 to 9999-12-31
TIME	Time in the form “23:59:59”, range -/+ 838:59:59
DATETIME	Combination of *DATE* and *TIME* in the form ‘2003-12-31 23:59:59’
YEAR(*2*|*4*)	A year in two or four digits

Table B.3: MySQL Built-in Date and Time Data Types

Appendix C Indexes

An index entry of the Index section of a book points to the location of the word/phrase indicated by the entry. The index entry helps you find the pages containing the word/phrase. Similarly, a column index of a table can speed up your finding data in a database. If your query has a condition on a column (or columns) that is (are) not indexed, the table will be fully scanned and the query will be slower than if an index was available.

This appendix shows you how to create various indexes. The topics covered are as follows.

- Creating an index
- Multi-column indexes
- Unique indexes
- Partial indexes
- Primary key index
- Deleting an index

Creating an Index

To create an index on a column or columns of a table, use this statement.

```
CREATE INDEX index ON table (columns);
```

For example, the statement in Listing C.1 creates an index named p_name_ix on the p_name column of the product table.

Listing C.1: Creating an index on the p_name in the product table

```
CREATE INDEX p_name_ix ON product (p_name);
```

Unique Index Names and Columns
You cannot have duplicate index names on a table. In addition, although you can, you should not create more than one index on a column(s) of a table, as MySQL future release will not allow you to create such indexes.

Multi-Column Indexes

An index can be based on multiple columns. A multi-column index is useful when you need to search on rows having the same value on an indexed column. For instance, if your query has to search on the **p_name** column of the **product** table and there can be more than one row with the same **p_name** but with different launch dates, it would help if you create an index on both **p_name** and **launch_dt** columns.

As an example, the statement in Listing C.2 creates a multi-column index on the **p_name** and **launch_dt** columns of the **product** table.

Listing C.2: Creating an index on multiple columns

```
CREATE INDEX p_name_launch_ix ON product (p_name, launch_dt);
```

Unique Indexes

In addition to speeding up your queries, a unique index also enforces unique values of the indexed column. A unique index can also be on multiple columns. Note that the syntax for creating a unique index contains a UNIQUE clause.

For example, the statement in Listing C.3 creates a unique index named **order_ibx** on the **c_order** table, on the **p_code** and **c_no** columns.

Listing C.3: Creating a bitmap index

```
CREATE UNIQUE INDEX order_uix ON c_order (p_code, c_no);
```

Partial Indexes

An index does not need to be on the whole column. An index that is not on a whole column is called a partial index. For example, the statement in Listing C.4 creates an index on the first three characters part of the **c_name** column.

Listing C.4: Creating a bitmap join index on the c_name column

```
CREATE INDEX p_name_pix ON customer(c_name(3));
```

Primary Key Indexes

In Chapter 1, "Storing and Maintaining Data", I discussed the need for a primary key for a table. When you create a table with a primary key, the column(s) you designate for the primary key will be uniquely indexed. In other words, a unique index is created based on the column(s).

You can also add a primary key to an existing table, the syntax of which is as follows.

```
ALTER TABLE table_name ADD PRIMARY KEY index_name(columns);
```

In Chapter 5, "Joins", I introduced the customer order table (**c_order**). Suppose when we created this table, you did not designate any primary key. You can create a primary using the statement in Listing C.5.

Listing C.5: Adding a Primary Key

```
ALTER TABLE c_order ADD PRIMARY KEY c_order_pk(c_no, p_code,
      order_dt);
```

Deleting An Index

To delete an index of any type, use the DROP INDEX statement. The syntax is as follows.

```
DROP index_name ON table_name;
```

For example, the statement in Listing C.6 deletes the **p_code_ix** index on the **product** table.

Listing C.6: Dropping an index

```
DROP INDEX p_name_ix ON product;
```

Index