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.