Chapter 3. Basic SQL

As mentioned in Chapter 2, Dr. Edgar F. Codd conceived the relational database model and its normal forms in the early 1970s. In 1974, researchers at IBM's San Jose lab began work on a major project intended to prove the relational model's viability, called System R. At the same time, Dr. Donald Chamberlin and his colleagues were also working to define a database language. They developed the Structured English Query Language (SEQUEL), which allowed users to query a relational database using clearly defined English-style sentences. This was later renamed Structured Query Language (SQL), for legal reasons.

The first database management systems based on SQL became available commercially by the end of the '70s. With the growing activity surrounding the development of database languages, standardization emerged to simplify things, and the community settled on SQL. Both the American and international standards organizations (ANSI and ISO) took part in the standardization process, and in 1986 the first SQL standard was approved. The standard was later revised several times, with the names (SQL:1999, SQL:2003, SQL:2008, etc.) indicating the versions released in the corresponding years. We will use the phrase *the SQL standard* or *standard SQL* to mean the current version of the SQL standard at any time.

MySQL extends the standard SQL, providing extra features. For example, MySQL implements the `STRAIGHT_JOIN`, which is syntax not recognized by other DBMSs.

This chapter introduces MySQL's SQL implementation, which we often refer to as the *CRUD* operations: `create`, `read`, `update`, and `delete`. We will show you how to read data from a database with the `SELECT` statement and choose what data to retrieve and in which order it is displayed. We'll also show you the basics of modifying your databases with the `INSERT` statement to add data, `UPDATE` to change data, and `DELETE` to remove data. Finally, we'll explain how to use the nonstandard `SHOW TABLES` and `SHOW COLUMNS` statements to explore your database.

Using the sakila Database

In Chapter 2, we showed you the principles of how to build a database diagram using the ER model. We also introduced the steps you take to convert an ER model to a format that makes sense for constructing a relational database. This section will show you the structure of the MySQL `sakila` database so you can start to get familiar with different database relational models. We won't explain the SQL statements used to create the database here; that's the subject of Chapter 4.

If you haven't imported the database yet, follow the steps in "Entity Relationship Modeling Examples" to perform the task.

To choose the `sakila` database as our current database, we will use the `USE` statement. Type the following command:

```
mysql> USE sakila;
```

```
Database changed
mysql>
```

You can check which is the active database by typing the **`SELECT DATABASE();`** command:

```
mysql> SELECT DATABASE();
```

```
+------------+
| DATABASE() |
+------------+
| sakila     |
+------------+
1 row in set (0.00 sec)
```

Now, let's explore what tables make up the `sakila` database using the `SHOW TABLES` statement:

```
mysql> SHOW TABLES;
```

```
+----------------------------+
| Tables_in_sakila           |
+----------------------------+
| actor                      |
| actor_info                 |
| ...                        |
| customer                   |
| customer_list              |
| film                       |
| film_actor                 |
| film_category              |
| film_list                  |
| film_text                  |
| inventory                  |
| language                   |
| nicer_but_slower_film_list |
| payment                    |
| rental                     |
| sales_by_film_category     |
| sales_by_store             |
| staff                      |
| staff_list                 |
| store                      |
+----------------------------+
23 rows in set (0.00 sec)
```

So far, there have been no surprises. Let's find out more about each of the tables that make up the `sakila` database. First, let's use the `SHOW COLUMNS` statement to explore the `actor` table (note that the output has been wrapped to fit with the page margins):

```
mysql> SHOW COLUMNS FROM actor;
```

```
+-------------+-------------------+------+-----+-------
| Field       | Type              | Null | Key | Defaul
+-------------+-------------------+------+-----+-------
| actor_id    | smallint unsigned | NO   | PRI | NULL
| first_name  | varchar(45)       | NO   |     | NULL
| last_name   | varchar(45)       | NO   | MUL | NULL
| last_update | timestamp         | NO   |     | CURREN
+-------------+-------------------+------+-----+-------
  ...+------------------------------------------+
 ...| Extra                                          |
 ...+------------------------------------------------+
 ...| auto_increment                                 |
 ...|                                                |
 ...|                                                |
 ...| DEFAULT_GENERATED on update CURRENT_TIMESTAMP  |
 ...+------------------------------------------------+
4 rows in set (0.01 sec)
```

The `DESCRIBE` keyword is identical to `SHOW COLUMNS FROM`, and we can abbreviate it to just `DESC`, so we can write the previous query as follows:

```
mysql> DESC actor;
```

The output produced is identical. Let's examine the table structure more closely. The `actor` table contains four columns, `actor_id`, `first_name`, `last_name`, and `last_update`. We can also extract the types of the columns: a `smallint` for `actor_id`, `varchar(45)` for `first_name` and `last_name`, and `timestamp` for `last_update`. None of the columns accepts `NULL` (empty) value, `actor_id` is the primary key (`PRI`), and `last_name` is the first column of a nonunique index (`MUL`). Don't worry about the details; all that's important right now are the column names we will use for the SQL commands.

Next let's explore the `city` table by executing the `DESC` statement:

```
mysql> DESC city;
```

```
+-------------+-------------------+------+-----+-------
| Field       | Type              | Null | Key | Defaul
+-------------+-------------------+------+-----+-------
| city_id     | smallint unsigned | NO   | PRI | NULL
| city        | varchar(50)       | NO   |     | NULL
| country_id  | smallint unsigned | NO   | MUL | NULL
| last_update | timestamp         | NO   |     | CURREN
+-------------+-------------------+------+-----+-------
...+------------------------------------------------+
...| Extra                                          |
...+------------------------------------------------+
...| auto_increment                                 |
...|                                                |
...|                                           |
...| DEFAULT_GENERATED on update CURRENT_TIMESTAMP |
...+-------------------------------------------------+
4 rows in set (0.01 sec)
```

NOTE

The DEFAULT_GENERATED that you see in the Extra column indicates that this particular column uses a default value. This information is a MySQL 8.0 notation particularity, and it is not present in MySQL 5.7 or MariaDB 10.5.

Again, what's important is getting familiar with the columns in each table, as we'll make frequent use of these later when we discuss querying.

The next section shows you how to explore the data that MySQL stores in the `sakila` database and its tables.

The SELECT Statement and Basic Querying Techniques

The previous chapters showed you how to install and configure MySQL and use the MySQL command line, and introduced the ER model. Now you're ready to start learning the SQL language that all MySQL clients use to explore and manipulate data. This section introduces the most commonly used SQL keyword: the `SELECT` keyword. We explain the fundamental elements of style and syntax and the features of the `WHERE` clause, Boolean operators, and sorting (much of this also applies to our later discussions of `INSERT`, `UPDATE`, and `DELETE`). This isn't the end of our discussion of `SELECT`; you'll find more in Chapter 5, where we show you how to use its advanced features.

Single-Table SELECTs

The most basic form of `SELECT` reads the data in all rows and columns from a table. Connect to MySQL using the command line and choose the `sakila` database:

```
mysql> USE sakila;
```

```
Database changed
```

Let's retrieve all of the data in the `language` table:

```
mysql> SELECT * FROM language;
```

```
+-------------+----------+---------------------+
| language_id | name     | last_update         |
+-------------+----------+---------------------+
|           1 | English  | 2006-02-15 05:02:19 |
|           2 | Italian  | 2006-02-15 05:02:19 |
|           3 | Japanese | 2006-02-15 05:02:19 |
|           4 | Mandarin | 2006-02-15 05:02:19 |
|           5 | French   | 2006-02-15 05:02:19 |
|           6 | German   | 2006-02-15 05:02:19 |
+-------------+----------+---------------------+
6 rows in set (0.00 sec)
```

The output has six rows, and each row contains the values for all the columns present in the table. We now know that there are six languages, and we can see the languages, their identifiers, and the last time each language was updated.

A simple `SELECT` statement has four components:

1. The keyword `SELECT` .
2. The columns to be displayed. The asterisk (`*`) symbol is a wildcard character meaning all columns.
3. The keyword `FROM` .
4. The table name.

So in this example, we've asked for all columns from the `language` table, and that's what MySQL has returned to us.

Let's try another simple `SELECT` . This time, we'll retrieve all columns from the `city` table:

```
mysql> SELECT * FROM city;
```

```
+---------+------------------------+-----------+------
| city_id | city                   | country_id | last_
+---------+------------------------+-----------+------
|       1 | A Corua (La Corua)     |         87 | 2006-
|       2 | Abha                   |         82 | 2006-
|       3 | Abu Dhabi              |        101 | 2006-
|     ...
|     599 | Zhoushan               |         23 | 2006-
|     600 | Ziguinchor             |         83 | 2006-
+---------+------------------------+-----------+------
600 rows in set (0.00 sec)
```

There are 600 cities, and the output has the same basic structure as in our first example.

This example provides some insight into how the relationships between the tables work. Consider the first row of the results. In the column `country_id`, you will see the value 87. As you'll see later, we can check the `country` table to find out that the country with code 87 is Spain. We'll discuss how to write queries on relationships between tables in "Joining Two Tables".

If you look at the complete output, you'll also see that there are several different cities with the same `country_id`. Having repeated `country_id` values isn't a problem since we expect a country to have many cities (a one-to-many relationship).

You should now feel comfortable choosing a database, listing its tables, and retrieving all of the data from a table using the `SELECT` statement. To practice, you might want to experiment with the other tables in the `sakila` database. Remember that you can use the `SHOW TABLES` statement to find out the table names.

Choosing Columns

Earlier, we used the `*` wildcard character to retrieve all the columns in a table. If you don't want to display all the columns, it's easy to be more specific by listing the columns you want, in the order you want them, separated by commas. For example, if you want only the `city` column from the `city` table, you'd type:

```
mysql> SELECT city FROM city;
```

```
+--------------------+
| city               |
+--------------------+
| A Corua (La Corua) |
| Abha               |
| Abu Dhabi          |
| Acua               |
| Adana              |
+--------------------+
5 rows in set (0.00 sec)
```

If you want both the `city` and `city_id` columns, in that order, you'd use:

```
mysql> SELECT city, city_id FROM city;
```

```
+--------------------+---------+
| city               | city_id |
+--------------------+---------+
| A Corua (La Corua) |       1 |
| Abha               |       2 |
| Abu Dhabi          |       3 |
| Acua               |       4 |
| Adana              |       5 |
+--------------------+---------+
5 rows in set (0.01 sec)
```

You can even list columns more than once:

```
mysql> SELECT city, city FROM city;
```

```
+--------------------+--------------------+
| city               | city               |
+--------------------+--------------------+
| A Corua (La Corua) | A Corua (La Corua) |
| Abha               | Abha               |
| Abu Dhabi          | Abu Dhabi          |
| Acua               | Acua               |
| Adana              | Adana              |
+--------------------+--------------------+
5 rows in set (0.00 sec)
```

Although this may seem pointless, it can be useful when combined with aliases in more advanced queries, as you'll see in Chapter 5.

You can specify database, table, and column names in a `SELECT` statement. This allows you to avoid the `USE` command and work with any database and table directly with `SELECT`; it also helps resolve ambiguities, as we'll show in "Joining Two Tables". For example, suppose you want to retrieve the `name` column from the `language` table in the `sakila` database. You can do this with the following command:

```
mysql> SELECT name FROM sakila.language;
```

```
+----------+
| name     |
+----------+
| English  |
| Italian  |
| Japanese |
| Mandarin |
| French   |
| German   |
+----------+
6 rows in set (0.01 sec)
```

The `sakila.language` component after the `FROM` keyword specifies the `sakila` database and its `language` table. There's no need to enter **`USE sakila;`** before running this query. This syntax can also be used with other SQL statements, including the `UPDATE`, `DELETE`, `INSERT`, and `SHOW` statements we discuss later in this chapter.

Selecting Rows with the WHERE Clause

This section introduces the `WHERE` clause and explains how to use operators to write expressions. You'll see these in `SELECT` statements and other statements such as `UPDATE` and `DELETE`; we'll show you examples later in this chapter.

WHERE basics

The `WHERE` clause is a powerful tool that allows you to filter which rows are returned from a `SELECT` statement. You use it to return rows that match a condition, such as having a column value that exactly matches a string, a number greater or less than a value, or a string that is a prefix of another. Almost all our examples in this and later chapters contain `WHERE` clauses, and you'll become very familiar with them.

The simplest `WHERE` clause is one that exactly matches a value. Consider an example where you want to find out the English language's details in the `language` table. Here's what you'd type:

```
mysql> SELECT * FROM sakila.language WHERE name = 'Engl
```

```
+-------------+---------+---------------------+
| language_id | name    | last_update         |
+-------------+---------+---------------------+
|           1 | English | 2006-02-15 05:02:19 |
+-------------+---------+---------------------+
1 row in set (0.00 sec)
```

MySQL returns all rows that match your search criteria—in this case, just the one row and all its columns.

Let's try another exact match example. Suppose you want to find out the first name of the actor with an `actor_id` value of 4 in the `actor` table. You would type:

```
mysql> SELECT first_name FROM actor WHERE actor_id = 4;
```

```
+------------+
| first_name |
+------------+
| JENNIFER   |
+------------+
1 row in set (0.00 sec)
```

Here you provide a column and a row, including the column `first_name` after the `SELECT` keyword and specifying the `WHERE actor_id = 4`.

If a value matches more than one row, the results will contain all the matches. Suppose you want to see all the cities belonging to Brazil, which has a `country_id` of 15. You would type in:

```
mysql> SELECT city FROM city WHERE country_id = 15;
```

```
+----------------------+
| city                 |
+----------------------+
| Alvorada             |
| Angra dos Reis       |
| Anpolis              |
| Aparecida de Goinia  |
| Araatuba             |
| Bag                  |
| Belm                 |
| Blumenau             |
| Boa Vista            |
| Braslia              |                |
| ...                  |
+----------------------+
28 rows in set (0.00 sec)
```

The results show the names of the 28 cities that belong to Brazil. If we could join the information we get from the `city` table with information we get from the `country` table, we could display the cities' names with their respective countries. We'll see how to perform this type of query in "Joining Two Tables".

Now let's retrieve values that belong to a range. Retrieving multiple values is simple for numeric ranges, so let's start by finding all cities' names with a `city_id` less than 5. To do this, execute the following statement:

```
mysql> SELECT city FROM city WHERE city_id < 5;
```

```
+--------------------+
| city               |
+--------------------+
| A Corua (La Corua) |
| Abha               |
| Abu Dhabi          |
| Acua               |
+--------------------+
4 rows in set (0.00 sec)
```

For numbers, the frequently used operators are equal (`=`), greater than (`>`), less than (`<`), less than or equal (`<=`), greater than or equal (`>=`), and not equal (`<>` or `!=`).

Consider one more example. If you want to find all languages that don't have a `language_id` of 2, you'd type:

```
mysql> SELECT language_id, name FROM sakila.language
    -> WHERE language_id <> 2;
```

```
+-------------+----------+
| language_id | name     |
+-------------+----------+
|           1 | English  |
|           3 | Japanese |
|           4 | Mandarin |
|           5 | French   |
|           6 | German   |
+-------------+----------+
5 rows in set (0.00 sec)
```

The previous output shows the first, third, and all subsequent languages in the table. Note that you can use either the `<>` or `!=` operator for the *not-equal* condition.

You can use the same operators for strings. By default, string comparisons are not case-sensitive and use the current character set. For example:

```
mysql> SELECT first_name FROM actor WHERE first_name <
```

```
+------------+
| first_name |
+------------+
| ALEC       |
| AUDREY     |
| ANNE       |
| ANGELA     |
| ADAM       |
| ANGELINA   |
| ALBERT     |
| ADAM       |
| ANGELA     |
| ALBERT     |
| AL         |
| ALAN       |
| AUDREY     |
+------------+
13 rows in set (0.00 sec)
```

By "not case-sensitive" we mean that `B` and `b` will be considered the same filter, so this query will provide the same result:

```
mysql> SELECT first_name FROM actor WHERE first_name <
```

```
+------------+
| first_name |
+------------+
| ALEC       |
| AUDREY     |
| ANNE       |
| ANGELA     |
| ADAM       |
| ANGELINA   |
| ALBERT     |
| ADAM       |
| ANGELA     |
| ALBERT     |
| AL         |
| ALAN       |
| AUDREY     |
+------------+
13 rows in set (0.00 sec)
```

Another common task to perform with strings is to find matches that begin with a prefix, contain a string, or end in a suffix. For example, we might want to find all album names beginning with the word “Retro.” We can do this with the `LIKE` operator in a `WHERE` clause. Let’s see an example where we are searching for a film with a title that contains the word `family`:

```
mysql> SELECT title FROM film WHERE title LIKE '%famil
```

```
+----------------+
| title          |
+----------------+
| CYCLONE FAMILY |
| DOGMA FAMILY   |
| FAMILY SWEET   |
+----------------+
3 rows in set (0.00 sec)
```

Let’s take a look at how this works. The `LIKE` clause is used with strings and means that a match must meet the pattern in the string that follows. In our example, we’ve used `LIKE '%family%'`, which means the string must contain `family`, and it can be preceded or followed by zero or more characters. Most strings used with `LIKE` contain the percentage character (`%`) as a wildcard character that matches all possible strings. You can use it to define a string that ends in a suffix—such as `"%ing"`—or a string that starts with a particular substring, such as `"Corruption%"`.

For example, `"John%"` would match all strings starting with `John`, such as `John Smith` and `John Paul Getty`. The pattern `"%Paul"` matches all strings that have `Paul` at the end. Finally, the pattern `"%Paul%"` matches all strings that have `Paul` in them, including at the start or at the end.

If you want to match exactly one wildcard character in a `LIKE` clause, you use the underscore character (`_`). For example, if you want the titles of all movies starring an actor whose name begins with the three letters NAT, you use:

```
mysql> SELECT title FROM film_list WHERE actors LIKE 'N
```

```
+-----------------------+
| title                 |
+-----------------------+
| FANTASY TROOPERS      |
| FOOL MOCKINGBIRD      |
| HOLES BRANNIGAN       |
| KWAI HOMEWARD         |
| LICENSE WEEKEND       |
| NETWORK PEAK          |
| NUTS TIES             |
| TWISTED PIRATES       |
| UNFORGIVEN ZOOLANDER |
+-----------------------+
9 rows in set (0.04 sec)
```

TIP

In general, you should avoid using the percentage (`%`) wildcard at the beginning of the pattern, like in the following example:

```
mysql> SELECT title FROM film WHERE title LIKE '%day%';
```

You will get the results, but MySQL will not use the index under this condition. Using the wildcard will force MySQL to read the entire table to retrieve the results, which can cause a severe performance impact if the table has millions of rows.

Combining conditions with AND, OR, NOT, and XOR

So far, we've used the `WHERE` clause to test one condition, returning all rows that meet it. You can combine two or more conditions using the Boolean operators `AND` , `OR` , `NOT` , and `XOR` .

Let's start with an example. Suppose you want to find the titles of sci-fi movies that are rated PG. This is straightforward with the `AND` operator:

```
mysql> SELECT title FROM film_list WHERE category LIKE
```

```
    -> AND rating LIKE 'PG';
```

```
+----------------------+
| title                |
+----------------------+
| CHAINSAW UPTOWN      |
| CHARADE DUFFEL       |
| FRISCO FORREST       |
| GOODFELLAS SALUTE    |
| GRAFFITI LOVE        |
| MOURNING PURPLE      |
| OPEN AFRICAN         |
| SILVERADO GOLDFINGER |
| TITANS JERK          |
| TROJAN TOMORROW      |
| UNFORGIVEN ZOOLANDER |
| WONDERLAND CHRISTMAS |
+----------------------+
12 rows in set (0.07 sec)
```

The AND operation in the WHERE clause restricts the results to those rows that meet both conditions.

The OR operator is used to find rows that meet at least one of several conditions. To illustrate, imagine now that you want a list of movies in the Children or Family categories. You can do this with OR and two LIKE clauses:

```
mysql> SELECT title FROM film_list WHERE category LIKE
    -> OR category LIKE 'Family';
```

```
+------------------------+
| title                  |
+------------------------+
| AFRICAN EGG            |
| APACHE DIVINE          |
| ATLANTIS CAUSE         |
...
| WRONG BEHAVIOR         |
| ZOOLANDER FICTION      |
+-------------------------+
129 rows in set (0.04 sec)
```

The OR operation in the WHERE clause restricts the answers to those that meet either of the two conditions. As an aside, we can observe that the results are ordered. This is merely a coincidence; in this case, they're reported in the order they were added to the database. We'll return to sorting output in "The ORDER BY Clause".

You can combine AND and OR , but you need to make it clear whether you want to first AND the conditions or OR them. Parentheses cluster parts of a statement together and help make expressions readable; you can use them just as you would in basic math. Let's say that now you want sci-fi or family movies that are rated PG. You can write this query as follows:

```
mysql> SELECT title FROM film_list WHERE (category like
    -> OR category LIKE 'Family') AND rating LIKE 'PG';
```

```
+-------------------------+
| title                   |
+-------------------------+
| BEDAZZLED MARRIED       |
| CHAINSAW UPTOWN         |
| CHARADE DUFFEL          |
| CHASING FIGHT           |
| EFFECT GLADIATOR        |
...
| UNFORGIVEN ZOOLANDER    |
| WONDERLAND CHRISTMAS    |
+-------------------------+
30 rows in set (0.07 sec)
```

The parentheses make the evaluation order clear: you want movies from either the Sci-Fi or the Family category, but all of them need to be PG-rated.

With the use of parentheses, it is possible to change the evaluation order. The easiest way to see how this works is by playing around with calculations:

```
mysql> SELECT (2+2)*3;
```

```
+---------+
| (2+2)*3 |
+---------+
|      12 |
+---------+
1 row in set (0.00 sec)


mysql> SELECT 2+2*3;


+-------+
| 2+2*3 |
+-------+
|     8 |
+-------+
1 row in set (0.00 sec)
```

NOTE

One of the most difficult problems to diagnose is a query that is running with no syntax errors, but it is returning values different from those expected. While the parentheses do not affect the `AND` operator, the `OR` operator is significantly impacted by them. For example, consider the result of this statement:

```
mysql> SELECT * FROM sakila.city WHERE city_id = 3
    -> OR city_id = 4 AND country_id = 60;
```

```
+---------+------------+------------+---------------------+
| city_id | city       | country_id | last_update         |
+---------+------------+------------+---------------------+
|       3 | Abu Dhabi  |        101 | 2006-02-15 04:45:25 |
|       4 | Acua       |         60 | 2006-02-15 04:45:25 |
+---------+------------+------------+---------------------+
2 rows in set (0.00 sec)
```

If we change the ordering of the operators, we will obtain a different result:

```
mysql> SELECT * FROM sakila.city WHERE country_id = 60
    -> AND city_id = 3 OR city_id = 4;
```

```
+---------+------+------------+---------------------+
| city_id | city | country_id | last_update         |
+---------+------+------------+---------------------+
|       4 | Acua |         60 | 2006-02-15 04:45:25 |
+---------+------+------------+---------------------+
1 row in set (0.00 sec)
```

Using parentheses makes the queries much easier to understand and increases the likelihood that you'll get the results you're expecting. We recommend that you use parentheses whenever there's a chance MySQL could misinterpret your intention; there's no good reason to rely on MySQL's implicit evaluation order.

The unary `NOT` operator negates a Boolean statement. Earlier we gave the example of listing all languages with a `language_id` not equal to 2. You can also write this query with the `NOT` operator:

```
mysql> SELECT language_id, name FROM sakila.language
```

```
    -> WHERE NOT (language_id = 2);
```

```
+-------------+----------+
| language_id | name     |
+-------------+----------+
|           1 | English  |
|           3 | Japanese |
|           4 | Mandarin |
|           5 | French   |
|           6 | German   |
+-------------+----------+
5 rows in set (0.01 sec)
```

The expression in parentheses, `(language_id = 2)`, gives the condition to match, and the `NOT` operation negates it, so you get everything but those results that match the condition. There are several other ways you can write a `WHERE` clause with the same idea. In Chapter 5, you will see that some have better performance than others.

Consider another example using `NOT` and parentheses. Suppose you want to get a list of all movie titles with an `FID` less than 7, but not those numbered 4 or 6. You can do this with the following query:

```
mysql> SELECT fid,title FROM film_list WHERE FID < 7 AN
```

```
+------+------------------+
| fid  | title            |
+------+------------------+
|    1 | ACADEMY DINOSAUR |
|    2 | ACE GOLDFINGER   |
|    3 | ADAPTATION HOLES |
|    5 | AFRICAN EGG      |
+------+------------------+
4 rows in set (0.06 sec)
```

Understanding operator precedence can be a little tricky, and sometimes it takes DBAs a long time to debug a query and identify why it is not returning the requested values. The following list shows the available operators in order from the highest priority to the lowest. Operators that are shown together on a line have the same priority:

- `INTERVAL`
- `BINARY`, `COLLATE`
- `!`
- `-` (unary minus), `~` (unary bit inversion)
- `^`
- `*`, `/`, `DIV`, `%`, `MOD`
- `-`, `+`
- `<<`, `>>`
- `&`
- `\|`
- `=` (comparison), `<=>`, `>=`, `>`, `<=`, `<`, `<>`, `!=`, `IS`, `LIKE`, `REGEXP`, `IN`, `MEMBER OF`
- `BETWEEN`, `CASE`, `WHEN`, `THEN`, `ELSE`
- `NOT`
- `AND`, `&&`
- `XOR`
- `OR`, `\|\|`
- `=` (assignment), `:=`

It is possible to combine these operators in diverse ways to get the desired results. For example, you can write a query to get the titles of any movies that have a price range between $2 and $4, belong to the Documentary or Horror category, and have an actor named Bob:

```
mysql> SELECT title
    -> FROM film_list
    -> WHERE price BETWEEN 2 AND 4
    -> AND (category LIKE 'Documentary' OR category LIK
    -> AND actors LIKE '%BOB%';
```

```
+------------------+
| title            |
+------------------+
| ADAPTATION HOLES |
+------------------+
1 row in set (0.08 sec)
```

Finally, before we move on to sorting, note that it is possible to execute queries that do not match any results. In this case, the query will return an empty set:

```
mysql> SELECT title FROM film_list
     -> WHERE price BETWEEN 2 AND 4
     -> AND (category LIKE 'Documentary' OR category LIK
     -> AND actors LIKE '%GRIPPA%';
```

```
Empty set (0.04 sec)
```

The ORDER BY Clause

We've discussed how to choose the columns and which rows are returned as part of the query result, but not how to control how the result is displayed. In a relational database, the rows in a table form a set; there is no intrinsic order between the rows, so we have to ask MySQL to sort the results if we want them in a particular order. This section explains how to use the `ORDER BY` clause to do this. Sorting does not affect *what* is returned; it only affects *what order* the results are returned in.

TIP

InnoDB tables in MySQL have a special index called the *clustered index* that stores row data. When you define a primary key on a table, InnoDB uses it as the clustered index. Suppose you are executing queries based on the primary key. In that case, the rows will be returned ordered in ascending order by the primary key. However, we always recommending using the `ORDER BY` clause if you want to enforce a particular order.

Suppose you want to return a list of the first 10 customers in the `sakila` database, sorted alphabetically by `name`. Here's what you'd type:

```
mysql> SELECT name FROM customer_list
     -> ORDER BY name
     -> LIMIT 10;
```

```
+--------------------+
| name               |
+--------------------+
| AARON SELBY        |
| ADAM GOOCH         |
| ADRIAN CLARY       |
| AGNES BISHOP       |
| ALAN KAHN          |
| ALBERT CROUSE      |
| ALBERTO HENNING    |
| ALEX GRESHAM       |
| ALEXANDER FENNELL  |
| ALFRED CASILLAS    |
+--------------------+
10 rows in set (0.01 sec)
```

The `ORDER BY` clause indicates that sorting is required, followed by the column that should be used as the sort key. In this example, you're sorting by name in alphabetically ascending order—the default sort is case-insensitive and in ascending order, and MySQL automatically sorts alphabetically because the columns are character strings. The way strings are sorted is determined by the character set and collation order that are being used. We discuss these in "Collation and Character Sets". For most of this book, we assume that you're using the default settings.

Let's look at another example. This time, you'll sort the output from the `address` table in ascending order based on the `last_update` column and show just the first five results:

```
mysql> SELECT address, last_update FROM address
    -> ORDER BY last_update LIMIT 5;
```

```
+-----------------------------+---------------------+
| address                     | last_update         |
+-----------------------------+---------------------+
| 1168 Najafabad Parkway      | 2014-09-25 22:29:59 |
| 1031 Daugavpils Parkway     | 2014-09-25 22:29:59 |
| 1924 Shimonoseki Drive      | 2014-09-25 22:29:59 |
| 757 Rustenburg Avenue       | 2014-09-25 22:30:01 |
| 1892 Nabereznyje Telny Lane | 2014-09-25 22:30:02 |
+-----------------------------+---------------------+
5 rows in set (0.00 sec)
```

As you can see, it is possible to sort different types of columns. Moreover, we can compound the sorting with two or more columns. For example, let’s say you want to sort the addresses alphabetically, but grouped by district:

```
mysql> SELECT address, district FROM address
    -> ORDER BY district, address;
```

```
+----------------------------------------+-------------
| address                                | district
+----------------------------------------+-------------
| 1368 Maracabo Boulevard                |
| 18 Duisburg Boulevard                  |
| 962 Tama Loop                          |
| 535 Ahmadnagar Manor                   | Abu Dhabi
| 669 Firozabad Loop                     | Abu Dhabi
| 1078 Stara Zagora Drive                | Aceh
| 663 Baha Blanca Parkway                | Adana
| 842 Salzburg Lane                      | Adana
| 614 Pak Kret Street                    | Addis Abeba
| 751 Lima Loop                          | Aden
| 1157 Nyeri Loop                        | Adygea
| 387 Mwene-Ditu Drive                   | Ahal
| 775 ostka Drive                        | al-Daqahliya
| ...
| 1416 San Juan Bautista Tuxtepec Avenue | Zufar
| 138 Caracas Boulevard                  | Zulia
+----------------------------------------+-------------
603 rows in set (0.00 sec)
```

You can also sort in descending order, and you can control this behavior for each sort key. Suppose you want to sort the addresses by descending alphabetical order and the districts in ascending order. You would type this:

```
mysql> SELECT address,district FROM address
    -> ORDER BY district ASC, address DESC
    -> LIMIT 10;
```

```
+-------------------------+-------------+
| address                 | district    |
+-------------------------+-------------+
| 962 Tama Loop           |             |
| 18 Duisburg Boulevard   |             |
| 1368 Maracabo Boulevard |             |
| 669 Firozabad Loop      | Abu Dhabi   |
| 535 Ahmadnagar Manor    | Abu Dhabi   |
| 1078 Stara Zagora Drive | Aceh        |
| 842 Salzburg Lane       | Adana       |
| 663 Baha Blanca Parkway | Adana       |
| 614 Pak Kret Street     | Addis Abeba |
| 751 Lima Loop           | Aden        |
+-------------------------+-------------+
10 rows in set (0.01 sec)
```

If a collision of values occurs and you don't specify another sort key, the sort order is undefined. This may not be important for you; you may not care about the order in which two customers with the identical name "John A. Smith" appear. If you want to enforce a certain order in this case, you need to add more columns to the ORDER BY clause, as demonstrated in the previous example.

The LIMIT Clause

As you may have noted, a few of the previous queries used the `LIMIT` clause. This is a useful nonstandard SQL statement that allows you to control how many rows are output. Its basic form allows you to limit the number of rows returned from a `SELECT` statement, which is useful when you want to restrict the amount of data communicated over a network or output to the screen. You might use it, for example, to get a sample of the data from a table, as shown here:

```
mysql> SELECT name FROM customer_list LIMIT 10;
```

```
+------------------+
| name             |
+------------------+
| VERA MCCOY       |
| MARIO CHEATHAM   |
| JUDY GRAY        |
| JUNE CARROLL      |
| ANTHONY SCHWAB    |
| CLAUDE HERZOG     |
| MARTIN BALES      |
| BOBBY BOUDREAU    |
| WILLIE MARKHAM    |
| JORDAN ARCHULETA |
+-------------------+
```

The `LIMIT` clause can have two arguments. In this case, the first argument specifies the first row to return, and the second specifies the maximum number of rows to return. The first argument is known as the *offset*. Suppose you want five rows, but you want to skip the first five rows, which means the result will start at the sixth row. Record offsets for `LIMIT` start at 0, so you can do this as follows:

```
mysql> SELECT name FROM customer_list LIMIT 5, 5;
```

```
+-------------------+
| name              |
+-------------------+
| CLAUDE HERZOG     |
| MARTIN BALES      |
| BOBBY BOUDREAU    |
| WILLIE MARKHAM    |
| JORDAN ARCHULETA |
+-------------------+
5 rows in set (0.00 sec)
```

The output is rows 6 to 10 from the `SELECT` query.

There's an alternative syntax that you might see for the `LIMIT` keyword: instead of writing `LIMIT 10, 5`, you can write `LIMIT 10 OFFSET 5`. The `OFFSET` syntax discards the *N* values specified in it.

Here's an example with no offset:

```
mysql> SELECT id, name FROM customer_list
    -> ORDER BY id LIMIT 10;
```

```
+----+------------------+
| ID | name             |
+----+------------------+
|  1 | MARY SMITH       |
|  2 | PATRICIA JOHNSON |
|  3 | LINDA WILLIAMS   |
|  4 | BARBARA JONES    |
|  5 | ELIZABETH BROWN  |
|  6 | JENNIFER DAVIS   |
|  7 | MARIA MILLER     |
|  8 | SUSAN WILSON     |
|  9 | MARGARET MOORE   |
| 10 | DOROTHY TAYLOR   |
+----+------------------+
10 rows in set (0.00 sec)
```

And here are the results with an offset of 5:

```
mysql> SELECT id, name FROM customer_list
    -> ORDER BY id LIMIT 10 OFFSET 5;
```

```
+----+----------------+
| ID | name           |
+----+----------------+
|  6 | JENNIFER DAVIS |
|  7 | MARIA MILLER   |
|  8 | SUSAN WILSON   |
|  9 | MARGARET MOORE |
| 10 | DOROTHY TAYLOR |
| 11 | LISA ANDERSON  |
| 12 | NANCY THOMAS   |
| 13 | KAREN JACKSON  |
| 14 | BETTY WHITE    |
| 15 | HELEN HARRIS   |
+----+----------------+
10 rows in set (0.01 sec)
```

Joining Two Tables

So far we've only been working with one table in our `SELECT` queries. However, the majority of cases will require information from more than one table at once. As we've explored the tables in the `sakila` database, it's become obvious that by using relationships, we can answer more interesting queries. For example, it'd be useful to know the country each city is in. This section shows you how to answer queries like that by joining two tables. We'll return to this issue as part of a longer, more advanced discussion of joins in Chapter 5.

We use only one join syntax in this chapter. There are two more (`LEFT` and `RIGHT JOIN`), and each gives you a different way to bring together data from two or more tables. The syntax we use here is the `INNER JOIN` , which is the most commonly used in daily activities. Let's look at an example, and then we'll explain more about how it works:

```
mysql> SELECT city, country FROM city INNER JOIN countr
    -> ON city.country_id = country.country_id
    -> WHERE country.country_id < 5
    -> ORDER BY country, city;
```

```
+----------+----------------+
| city     | country        |
+----------+----------------+
| Kabul    | Afghanistan    |
| Batna    | Algeria        |
| Bchar    | Algeria        |
| Skikda   | Algeria        |
| Tafuna   | American Samoa |
| Benguela | Angola         |
| Namibe   | Angola         |
+----------+----------------+
7 rows in set (0.00 sec)
```

The output shows the cities in each country with a `country_id` lower than 5. You can see for the first time which cities are in each country.

How does the `INNER JOIN` work? The statement has two parts: first, two table names separated by the `INNER JOIN` keywords; and second, the `ON` keyword that specifies the required columns to compose the condition. In this example, the two tables to be joined are `city` and `country` , expressed as `city INNER JOIN country` (for the basic `INNER JOIN` , it doesn't matter what order you list the tables in, so using `country INNER JOIN city` would have the same effect). The `ON` clause (`ON city.country_id`

`= country.country_id`) is where we tell MySQL the columns that hold the relationship between the tables; you should recall this from our design and our previous discussion in Chapter 2.

If in the join condition the column names in both tables used for matching are the same, you can use the `USING` clause instead:

```
mysql> SELECT city, country FROM city
    -> INNER JOIN country using (country_id)
    -> WHERE country.country_id < 5
    -> ORDER BY country, city;
```

```
+----------+----------------+
| city     | country        |
+----------+----------------+
| Kabul    | Afghanistan    |
| Batna    | Algeria        |
| Bchar    | Algeria        |
| Skikda   | Algeria        |
| Tafuna   | American Samoa |
| Benguela | Angola         |
| Namibe   | Angola         |
+----------+----------------+
7 rows in set (0.01 sec)
```

The Venn diagram in Figure 3-1 illustrates the inner join.

Before we leave `SELECT` , we'll give you a taste of one of the functions you can use to aggregate values. Suppose you want to count how many cities Italy has in our database. You can do this by joining the two tables and counting the number of rows with that `country_id` . Here's how it works:

```
mysql> SELECT COUNT(1) FROM city INNER JOIN country
    -> ON city.country_id = country.country_id
    -> WHERE country.country_id = 49
    -> ORDER BY country, city;
```

```
+----------+
| count(1) |
+----------+
|        7 |
+----------+
1 row in set (0.00 sec)
```

We explain more features of `SELECT` and aggregate functions in Chapter 5. For more on the `COUNT()` function, see "Aggregate functions".

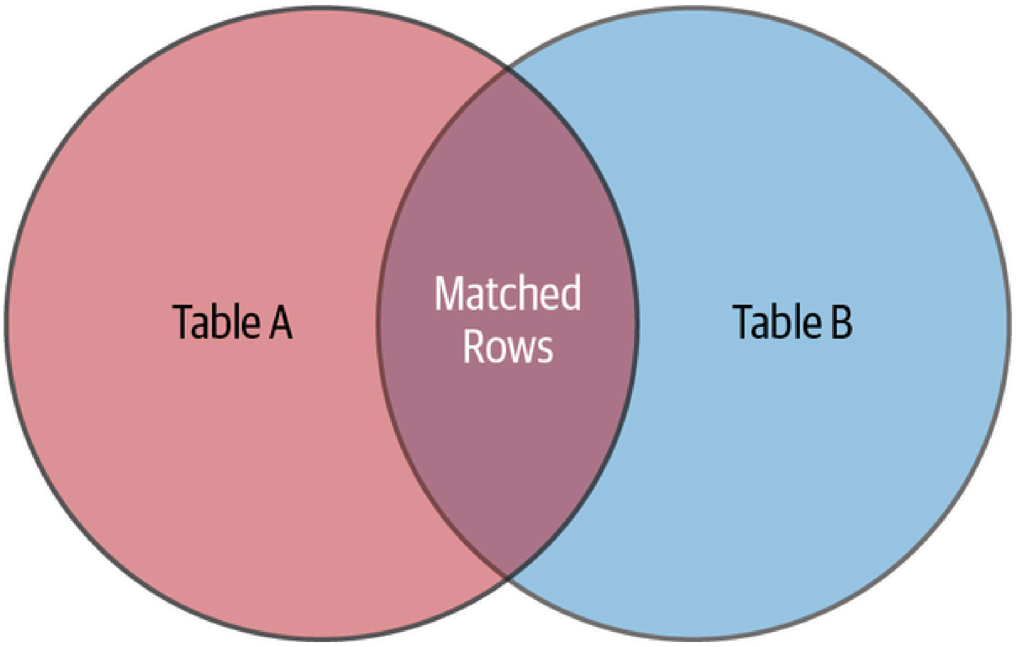

Figure 3-1. The Venn diagram representation of the INNER JOIN

The INSERT Statement

The `INSERT` statement is used to add new data to tables. This section explains its basic syntax and walks through some simple examples that add new rows to the `sakila` database. In Chapter 4, we'll discuss how to load data from existing tables or external data sources.

INSERT Basics

Inserting data typically occurs in two situations: when you bulk-load in a large batch as you create your database, and when you add data on an ad hoc basis as you use the database. In MySQL, different optimizations are built into the server for each situation. Importantly, different SQL syntaxes are available to make it easy for you to work with the server in both cases. We'll explain the basic `INSERT` syntax in this section and show you examples of using it for bulk and single-record insertion.

Let's start with the basic task of inserting one new row into the `language` table. To do this, you need to understand the table's structure. As we explained in “Using the sakila Database”, you can discover this with the `SHOW COLUMNS` statement:

```
mysql> SHOW COLUMNS FROM language;
```

```
+-------------+-------------------+------+-----+-------
| Field       | Type              | Null | Key | Default
+-------------+-------------------+------+-----+-------
| language_id | tinyint unsigned  | NO   | PRI | NULL
| name        | char(20)          | NO   |     | NULL
| last_update | timestamp         | NO   |     | CURRENT
+-------------+-------------------+------+-----+-------
...+-----------------------------------------------+
...| Extra                                         |
...+-----------------------------------------------+
...| auto_increment                                |
...|                                               |
...| DEFAULT_GENERATED on update CURRENT_TIMESTAMP |
...+-----------------------------------------------+
3 rows in set (0.00 sec)
```

This tells you that the `language_id` column is auto-generated, and the `last_update` column is updated every time an `UPDATE` operation happens. You’ll learn more about the `AUTO_INCREMENT` shortcut to automatically assign the next available identifier in Chapter 4.

Let’s add a new row for the language Portuguese. There are two ways to do this. The most common is to let MySQL fill in the default value for the `language_id`, like this:

```
mysql> INSERT INTO language VALUES (NULL, 'Portuguese',
```

```
Query OK, 1 row affected (0.10 sec)
```

If you we execute a `SELECT` on the table now, we’ll see that MySQL inserted the row:

```
mysql> SELECT * FROM language;
```

```
+-------------+------------+---------------------+
| language_id | name       | last_update         |
+-------------+------------+---------------------+
|           1 | English    | 2006-02-15 05:02:19 |
|           2 | Italian    | 2006-02-15 05:02:19 |
|           3 | Japanese   | 2006-02-15 05:02:19 |
|           4 | Mandarin   | 2006-02-15 05:02:19 |
|           5 | French     | 2006-02-15 05:02:19 |
|           6 | German     | 2006-02-15 05:02:19 |
|           7 | Portuguese | 2020-09-26 09:11:36 |
+-------------+------------+---------------------+
7 rows in set (0.00 sec)
```

Note that we used the function `NOW()` in the `last_update` column. The `NOW()` function returns the current date and time of the MySQL server.

The second option is to insert the value of the `language_id` column manually. Now that we already have seven languages, we should use 8 for the next value of the `language_id` . We can verify that with this SQL instruction:

```
mysql> SELECT MAX(language_id) FROM language;
```

```
+------------------+
| max(language_id) |
+------------------+
|                7 |
+------------------+
1 row in set (0.00 sec)
```

The `MAX()` function tells you the maximum value for the column supplied as a parameter. This is cleaner than using `SELECT language_id FROM language` , which prints out all the rows and requires you to inspect them to find the maximum value. Adding an `ORDER BY` and a `LIMIT` clause makes this easier, but using `MAX()` is much simpler than `SELECT language_id FROM language ORDER BY language_id DESC LIMIT 1` , which returns the same answer.

We’re now ready to insert the row. In this `INSERT`, we are going to insert the `last_update` value manually too. Here’s the needed command:

```
mysql> INSERT INTO language VALUES (8, 'Russian', '202
```

```
Query OK, 1 row affected (0.02 sec)
```

MySQL reports that one row has been affected (added, in this case), which we can confirm by checking the contents of the table again:

```
mysql> SELECT * FROM language;
```

```
+-------------+------------+---------------------+
| language_id | name       | last_update         |
+-------------+------------+---------------------+
|           1 | English    | 2006-02-15 05:02:19 |
|           2 | Italian    | 2006-02-15 05:02:19 |
|           3 | Japanese   | 2006-02-15 05:02:19 |
|           4 | Mandarin   | 2006-02-15 05:02:19 |
|           5 | French     | 2006-02-15 05:02:19 |
|           6 | German     | 2006-02-15 05:02:19 |
|           7 | Portuguese | 2020-09-26 09:11:36 |
|           8 | Russian    | 2020-09-26 10:35:00 |
+-------------+------------+---------------------+
8 rows in set (0.00 sec)
```

The single-row `INSERT` style detects primary key duplicates and stops as soon as it finds one. For example, suppose we try to insert another row with the same `language_id`:

```
mysql> INSERT INTO language VALUES (8, 'Arabic', '2020-
```

```
ERROR 1062 (23000): Duplicate entry '8' for key 'langua
```

The `INSERT` operation stops when it detects the duplicate key. You can add an `IGNORE` clause to prevent the error if you want, but note that the row still will not be inserted:

```
mysql> INSERT IGNORE INTO language VALUES (8, 'Arabic',
```

```
Query OK, 0 rows affected, 1 warning (0.00 sec)
```

In most cases you'll want to know about possible problems, though (after all, primary keys are supposed to be unique), so this `IGNORE` syntax is rarely used.

It is also possible to insert multiple values at once:

```
mysql> INSERT INTO language VALUES (NULL, 'Spanish', NC
    -> (NULL, 'Hebrew', NOW());
```

```
Query OK, 2 rows affected (0.02 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

Note that MySQL reports the results of bulk insertion differently from single insertion.

The first line tells you how many rows were inserted, while the first entry in the second line tells you how many rows (or records) were actually processed. If you use `INSERT IGNORE` and try to insert a duplicate record (one for which the primary key matches that of an existing row), MySQL will quietly skip inserting it and report it as a duplicate in the second entry on the second line:

```
mysql> INSERT IGNORE INTO language VALUES (9, 'Portugue
    (11, 'Hebrew', NOW());
```

```
Query OK, 1 row affected, 1 warning (0.01 sec)
Records: 2  Duplicates: 1  Warnings: 1
```

We discuss the causes of warnings, shown as the third entry on the second line of output, in Chapter 4.

Alternative Syntaxes

There are several alternatives to the `VALUES` syntax demonstrated in the previous section. This section walks through them and explains the advantages and drawbacks of each. If you're happy with the basic syntax we've described so far and want to move on to a new topic, feel free to skip ahead to "The DELETE Statement".

There are some advantages to the `VALUES` syntax we've been using: it works for both single and bulk inserts, you get an error message if you forget to supply values for all the columns, and you don't have to type in the column names. However, it also has some disadvantages: you need to remember the order of the columns, you need to provide a value for each column, and the syntax is closely tied to the underlying table structure. That is, if you change the table's structure, you need to change the `INSERT` statements. Fortunately, we can avoid these disadvantages by varying the syntax.

Suppose you know that the `actor` table has four columns, and you recall their names, but you've forgotten their order. You can insert a row using the following approach:

```
mysql> INSERT INTO actor (actor_id, first_name, last_na
    -> VALUES (NULL, 'Vinicius', 'Grippa', NOW());
```

```
Query OK, 1 row affected (0.03 sec)
```

The column names are included in parentheses after the table name, and the values stored in those columns are listed in parentheses after the `VALUES` keyword. So, in this example, a new row is created, and the value `201` is stored as the `actor_id` (remember, `actor_id` has the `auto_increment` property), `Vinicius` is stored as the `first_name`, `Grippa` is stored as the `last_name`, and the `last_update` column is populated with the current timestamp. This syntax's advantages are that it's readable and flexible (addressing the third disadvantage we described) and order-independent (addressing the first disadvantage). The burden is that you need to know the column names and type them in.

This new syntax can also address the second disadvantage of the simpler approach—that is, it can allow you to insert values for only some columns. To understand how this might be useful, let's explore the `city` table:

```
mysql> DESC city;
```

```
+-------------+----------------------+------+-----+----
| Field       | Type                 | Null | Key | Def
+-------------+----------------------+------+-----+----
| city_id     | smallint(5) unsigned | NO   | PRI | NUL
| city        | varchar(50)          | NO   |     | NUL
| country_id  | smallint(5) unsigned | NO   | MUL | NUL
| last_update | timestamp            | NO   |     | CUF
+-------------+----------------------+------+-----+----
...+--------------------------------------------+
...| Extra                                      |
...+--------------------------------------------+
...| auto_increment                             |
...|                                            |
...|                                            |
...| on update CURRENT_TIMESTAMP                |
...|--------------------------------------------+

4 rows in set (0.00 sec)
```

Notice that the `last_update` column has a default value of `CURRENT_TIMESTAMP`. This means that if you don't insert a value for the `last_update` column, MySQL will insert the current date and time by default. This is just what we want: when we store a record, we don't want to bother checking the date and time and typing it in. Let's try inserting an incomplete entry:

```
mysql> INSERT INTO city (city, country_id) VALUES ('Bel
```

```
Query OK, 1 row affected (0.00 sec)
```

We didn't set a value for the `city_id` column, so MySQL defaults it to the next available value (because of the `auto_increment` property), and `last_update` stores the current date and time. You can check this with a query:

```
mysql> SELECT * FROM city where city like 'Bebedouro';
```

```
+---------+-----------+------------+------------------
| city_id | city      | country_id | last_update
+---------+-----------+------------+------------------
|     601 | Bebedouro |         19 | 2021-02-27 21:34:6
+---------+-----------+------------+------------------
1 row in set (0.01 sec)
```

You can also use this approach for bulk insertion, as follows:

```
mysql> INSERT INTO city (city,country_id) VALUES
    -> ('Sao Carlos',19),
    -> ('Araraquara',19),
    -> ('Ribeirao Preto',19);
```

```
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

In addition to needing to remember and type in column names, a disadvantage of this approach is that you can accidentally omit values for columns. MySQL will set the omitted columns to the default values. All columns in a MySQL table have a default value of `NULL`, unless another default value is explicitly assigned when the table is created or modified.

When you need to use default values for the table columns, you might want to use the `DEFAULT` keyword (supported by MySQL 5.7 and later). Here's an example that adds a row to the `country` table using `DEFAULT`:

```
mysql> INSERT INTO country VALUES (NULL, 'Uruguay', DEF
```

```
Query OK, 1 row affected (0.01 sec)
```

The keyword DEFAULT tells MySQL to use the default value for that column, so the current date and time are inserted in our example. This approach's advantages are that you can use the bulk-insert feature with default values, and you can never accidentally omit a column.

There's another alternative INSERT syntax. In this approach, you list the column names and values together, so you don't have to mentally map the list of values to the earlier list of columns. Here's an example that adds a new row to the country table:

```
mysql> INSERT INTO country SET country_id=NULL,
    -> country='Bahamas', last_update=NOW();


Query OK, 1 row affected (0.01 sec)
```

The syntax requires you to list a table name, the keyword SET, and then column-equals-value pairs, separated by commas. Columns for which values aren't supplied are set to their default values. Again, the disadvantages are that you can accidentally omit values for columns and that you need to remember and type in column names. A significant additional disadvantage is that you can't use this method for bulk insertion.

You can also insert using values returned from a query. We discuss this in Chapter 7.

The DELETE Statement

The DELETE statement is used to remove one or more rows from a table. We explain single-table deletes here and discuss multitable deletes—which remove data from two or more tables through one statement—in Chapter 7.

DELETE Basics

The simplest use of DELETE is to remove all the rows in a table. Suppose you want to empty your rental table. You can do this with:

```
mysql> DELETE FROM rental;
```

```
Query OK, 16044 rows affected (2.41 sec)
```

The DELETE syntax doesn't include column names since it's used to remove whole rows and not just values from a row. To reset or modify a value in a row, you use the UPDATE statement, described in "The UPDATE Statement". Note that the DELETE statement doesn't remove the table itself. For example, having deleted all the rows in the rental table, you can still query the table:

```
mysql> SELECT * FROM rental;
```

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

You can also continue to explore its structure using DESCRIBE or SHOW CREATE TABLE, and insert new rows using INSERT. To remove a table, you use the DROP statement described in Chapter 4.

Note that if the table has a relationship with another table, the delete might fail because of the foreign key constraint:

```
mysql> DELETE FROM language;
```

```
ERROR 1451 (23000): Cannot delete or update a parent rc
constraint fails (`sakila`.`film`, CONSTRAINT `fk_film_
(`language_id`) REFERENCES `language` (`language_id`) C
```

Using WHERE, ORDER BY, and LIMIT

If you deleted rows in the previous section, reload your sakila database now by following the instructions in "Entity Relationship Modeling Examples". You'll need the rows in the rental table restored for the examples in this section.

To remove one or more rows, but not all rows in a table, use a WHERE clause. This works in the same way as it does for SELECT. For example, suppose you want to remove all rows from the rental table with a rental_id less than 10. You can do this with:

```
mysql> DELETE FROM rental WHERE rental_id < 10;
```

```
Query OK, 9 rows affected (0.01 sec)
```

The result is that the nine rows that match the criterion are removed.

Now suppose you want to remove all the payments from a customer called Mary Smith from the database. First, perform a `SELECT` with the `customer` and `payment` tables using `INNER JOIN` (as described in "Joining Two Tables"):

```
mysql> SELECT first_name, last_name, customer.customer_
    -> amount, payment_date FROM payment INNER JOIN cus
    -> ON customer.customer_id=payment.customer_id
    -> WHERE first_name like 'Mary'
    -> AND last_name like 'Smith';
```

```
+------------+-----------+-------------+--------+------
| first_name | last_name | customer_id | amount | payme
+------------+-----------+-------------+--------+------
| MARY       | SMITH     |           1 |   2.99 | 2005-
| MARY       | SMITH     |           1 |   0.99 | 2005-
| MARY       | SMITH     |           1 |   5.99 | 2005-
| MARY       | SMITH     |           1 |   0.99 | 2005-
...
| MARY       | SMITH     |           1 |   1.99 | 2005-
| MARY       | SMITH     |           1 |   2.99 | 2005-
| MARY       | SMITH     |           1 |   5.99 | 2005-
+------------+-----------+-------------+--------+------
32 rows in set (0.00 sec)
```

Next, perform the following `DELETE` operation to remove the row with a *customer_id* of 1 from the `payment` table:

```
mysql> DELETE FROM payment where customer_id=1;
```

```
Query OK, 32 rows affected (0.01 sec)
```

You can use the `ORDER BY` and `LIMIT` clauses with `DELETE`. You usually do this when you want to limit the number of rows deleted. For example:

```
mysql> DELETE FROM payment ORDER BY customer_id LIMIT 1
```

```
Query OK, 10000 rows affected (0.22 sec)
```

TIP

We highly recommend using `DELETE` and `UPDATE` operations for small sets of rows, due to performance issues. The appropriate value varies depending on the hardware, but a good rule of thumb is around 20,000–40,000 rows per batch.

Removing All Rows with TRUNCATE

If you want to remove all the rows in a table, there's a faster method than removing them with `DELETE`. When you use the `TRUNCATE TABLE` statement, MySQL takes the shortcut of dropping the table, removing the table structures, and then re-creating them. When there are many rows in a table, this is much faster.

NOTE

As a curiosity, there is a bug in MySQL 5.6 that can cause it to stall MySQL when performing a `TRUNCATE` operation when MySQL is configured with a large InnoDB buffer pool (200 GB or more). See the bug report for details.

If you want to remove all the data in the `payment` table, you can execute this:

```
mysql> TRUNCATE TABLE payment;
```

```
Query OK, 0 rows affected (0.07 sec)
```

Notice that the number of rows affected is shown as zero: to speed up the operation, MySQL doesn't count the number of rows that are deleted, so the number shown does not reflect the actual number of rows deleted.

The `TRUNCATE TABLE` statement differs from `DELETE` in a lot of ways, but it is worth mentioning a few:

- `TRUNCATE` operations drop and re-create the table, which is much faster than deleting rows one by one, particularly for large tables.
- `TRUNCATE` operations cause an implicit commit, so you can't roll them back.
- You cannot perform `TRUNCATE` operations if the session holds an active table lock.

Table types, transactions, and locking are discussed in Chapter 5. None of these limitations affects most applications in practice, and you can use `TRUNCATE TABLE` to speed up your processing. Of course, it's not common to delete whole tables during regular operation. An exception is temporary tables used to store query results for a particular user session temporarily, which can be deleted without losing the original data.

The UPDATE Statement

The `UPDATE` statement is used to change data. In this section, we show you how to update one or more rows in a single table. Multitable updates are discussed in "Updates".

If you've deleted rows from your `sakila` database, reload it before continuing.

Examples

The simplest use of the `UPDATE` statement is to change all the rows in a table. Suppose you need to update the `amount` column of the `payment` table by adding 10% for all payments. You could do this by executing:

```
mysql> UPDATE payment SET amount=amount*1.1;
```

```
Query OK, 16025 rows affected, 16025 warnings (0.41 sec
Rows matched: 16049  Changed: 16025  Warnings: 16025
```

Note that we forgot to update the `last_update` status. To make it coherent with the expected database model, you can fix this by running the following statement:

```
mysql> UPDATE payment SET last_update='2021-02-28 17:53
```

```
Query OK, 16049 rows affected (0.27 sec)
Rows matched: 16049  Changed: 16049  Warnings: 0
```

TIP

You can use the `NOW()` function to update the `last_update` column with the current timestamp of the execution. For example:

```
mysql> UPDATE payment SET last_update=NOW();
```

The second row reported by an `UPDATE` statement shows the overall effect of the statement. In our example, you see:

```
Rows matched: 16049  Changed: 16049  Warnings: 0
```

The first column reports the number of rows that were retrieved as matches; in this case, since there's no `WHERE` or `LIMIT` clause, all rows in the table match the query. The second column reports how many rows needed to be changed, which is always equal to or less than the number of rows that match. If you repeat the statement, you'll see a different result:

```
mysql> UPDATE payment SET last_update='2021-02-28 17:53
```

```
Query OK, 0 rows affected (0.07 sec)
Rows matched: 16049  Changed: 0  Warnings: 0
```

This time, since the date is already set to `2021-02-28 17:53:00` and there is no `WHERE` condition, all the rows still match the query but none are changed. Note also the number of rows changed is always equal to the number of rows affected, as reported on the first line of the output.

Using WHERE, ORDER BY, and LIMIT

Often, you don't want to change all the rows in a table. Instead, you want to update one or more rows that match a condition. As with `SELECT` and `DELETE`, the `WHERE` clause is used for the task. In addition, in the same way as with `DELETE`, you can use `ORDER BY` and `LIMIT` together to control how many rows are updated from an ordered list.

Let's try an example that modifies one row in a table. Suppose that the actress Penelope Guiness has changed her last name. To update it in the `actor` table of the database, you need to execute:

```
mysql> UPDATE actor SET last_name= UPPER('cruz')
    -> WHERE first_name LIKE 'PENELOPE'
    -> AND last_name LIKE 'GUINESS';
```

```
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

As expected, MySQL matched one row and changed one row.

To control how many updates occur, you can use the combination of `ORDER BY` and `LIMIT`:

```
mysql> UPDATE payment SET last_update=NOW() LIMIT 10;
```

```
Query OK, 10 rows affected (0.01 sec)
Rows matched: 10  Changed: 10  Warnings: 0
```

As with `DELETE`, you would do this because you either want to perform the operation in small chunks or modify only some rows. Here, you can see that 10 rows were matched and changed.

The previous query also illustrates an important aspect of updates. As you've seen, updates have two phases: a matching phase, where rows are found that match the `WHERE` clause, and a modification phase, where the rows that need changing are updated.

Exploring Databases and Tables with SHOW and mysqlshow

We've already explained how you can use the `SHOW` command to obtain information on the structure of a database, its tables, and the table columns. In this section, we'll review the most common types of the `SHOW` statement with brief examples using the `sakila` database. The `mysqlshow` command-line program performs the same functions as several `SHOW` command variants, but without you needing to start the MySQL client.

The `SHOW DATABASES` statement lists the databases you can access. If you've followed our sample database installation steps in "Entity Relationship Modeling Examples" and deployed the bank model in "Creating a Bank Database ER Model", your output should be as follows:

```
mysql> SHOW DATABASES;
```

```
+--------------------+
| Database           |
+--------------------+
| information_schema |
| bank_model         |
| employees          |
| mysql              |
| performance_schema |
| sakila             |
| sys                |
| world              |
+--------------------+
8 rows in set (0.01 sec)
```

These are the databases that you can access with the `USE` command (discussed in Chapter 4); if you have access privileges for other databases on your server, these will be listed too. You can only see databases for which you have some privileges, unless you have the global `SHOW DATABASES` privilege. You can get the same effect from the command line using the `mysqlshow` program:

```
$ mysqlshow -uroot -pmsandbox -h 127.0.0.1 -P 3306
```

You can add a `LIKE` clause to `SHOW DATABASES`. This is useful if you have many databases and want a short list as output. For example, to see only databases whose names begin with `s`, run:

```
mysql> SHOW DATABASES LIKE 's%';
```

```
+---------------+
| Database (s%) |
+---------------+
| sakila        |
| sys           |
+---------------+
2 rows in set (0.00 sec)
```

The `LIKE` statement's syntax is identical to its use in `SELECT`.

To see the statement used to create a database, you can use the `SHOW CREATE DATABASE` statement. For example, to see how you created `sakila`, type:

```
mysql> SHOW CREATE DATABASE sakila;
```

```
************************** 1. row *********************
       Database: sakila
Create Database: CREATE DATABASE `sakila` /*!40100 DEFA
utf8mb4 COLLATE utf8mb4_0900_ai_ci */ /*!80016 DEFAULT
1 row in set (0.00 sec)
```

This is perhaps the least exciting `SHOW` statement; it only displays the statement. Note, though, that some additional comments are included, `/*!` and `*/`:

```
40100 DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_09(
80016 DEFAULT ENCRYPTION='N'
```

These comments contain MySQL-specific keywords that provide instructions that are unlikely to be understood by other database programs. A database server other than MySQL will ignore this comment text, so the syntax is usable by both MySQL and other database server software. The optional number at the start of the comment indicates the minimum version of MySQL that can process this particular instruction (for example, `40100` indicates version 4.01.00); older versions of MySQL ignore such instructions. You'll learn about creating databases in Chapter 4.

The `SHOW TABLES` statement lists the tables in a database. To check the tables in `sakila`, type:

```
mysql> SHOW TABLES FROM sakila;
```

```
+----------------------------+
| Tables_in_sakila           |
+----------------------------+
| actor                      |
| actor_info                 |
| address                    |
| category                   |
| city                       |
| country                    |
| customer                   |
| customer_list              |
| film                       |
| film_actor                 |
| film_category              |
| film_list                  |
| film_text                  |
| inventory                  |
| language                   |
| nicer_but_slower_film_list |
| payment                    |
| rental                     |
| sales_by_film_category     |
| sales_by_store             |
| staff                      |
| ...                     |
+----------------------------+
23 rows in set (0.01 sec)
```

If you've already selected the `sakila` database with the `USE sakila` command, you can use the shortcut:

```
mysql> SHOW TABLES;
```

You can get a similar result by specifying the database name to the `mysqlshow` program:

```
$ mysqlshow -uroot -pmsandbox -h 127.0.0.1 -P 3306 saki
```

As with `SHOW DATABASES`, you can't see tables that you don't have privileges for. This means you can't see tables in a database you can't access, even if you have the `SHOW DATABASES` global privilege.

The `SHOW COLUMNS` statement lists the columns in a table. For example, to check the columns of `country`, type:

```
mysql> SHOW COLUMNS FROM country;
```

```
*************************** 1. row *******************
  Field: country_id
   Type: smallint unsigned
   Null: NO
    Key: PRI
Default: NULL
  Extra: auto_increment
*************************** 2. row *******************
  Field: country
   Type: varchar(50)
   Null: NO
    Key:
Default: NULL
  Extra:
*************************** 3. row *******************
  Field: last_update
   Type: timestamp
   Null: NO
```

```
    Key:
Default: CURRENT_TIMESTAMP
  Extra: DEFAULT_GENERATED on update CURRENT_TIMESTAMP
3 rows in set (0.00 sec)
```

The output reports the names of all the columns, their types and sizes, whether they can be `NULL`, whether they are part of a key, their default values, and any extra information. Types, keys, `NULL` values, and defaults are discussed further in Chapter 4. If you haven't already chosen the `sakila` database with the `USE` command, then you can add the database name before the table name, as in `sakila.country`. Unlike with the previous `SHOW` statements, you can always see all column names if you have access to a table; it doesn't matter that you don't have certain privileges for all columns.

You can get a similar result by using `mysqlshow` with the database and table name:

```
$ mysqlshow -uroot -pmsandbox -h 127.0.0.1 -P 3306 saki
```

You can see the statement used to create a particular table using the `SHOW CREATE TABLE` statement (we'll also look at creating tables in Chapter 4). Some users prefer this output to that of `SHOW COLUMNS`, since it has the familiar format of a `CREATE TABLE` statement. Here's an example for the `country` table:

```
mysql> SHOW CREATE TABLE country\G
```

```
*************************** 1. row ********************
       Table: country
Create Table: CREATE TABLE `country` (
  `country_id` smallint unsigned NOT NULL AUTO_INCREMEN
  `country` varchar(50) NOT NULL,
  `last_update` timestamp NOT NULL DEFAULT CURRENT_TIME
  CURRENT_TIMESTAMP,
  PRIMARY KEY (`country_id`)
) ENGINE=InnoDB AUTO_INCREMENT=110 DEFAULT CHARSET=utf8
  COLLATE=utf8mb4_0900_ai_ci
1 row in set (0.00 sec)
```