Chapter 3. Query Primer

So far, you have seen a few examples of database queries (a.k.a. `select` statements) sprinkled throughout the first two chapters. Now it's time to take a closer look at the different parts of the `select` statement and how they interact. After finishing this chapter, you should have a basic understanding of how data is retrieved, joined, filtered, grouped, and sorted; these topics will be covered in detail in Chapters 4 through 10.

Query Mechanics

Before dissecting the `select` statement, it might be interesting to look at how

then you have already logged in to the MySQL server by providing your username and password (and possibly a hostname if the MySQL server is running on a different computer). Once the server has verified that your username and password are correct, a *database connection* is generated for you to use. This connection is held by the application that requested it (which, in this case, is the `mysql` tool) until the application releases the connection (i.e., as a result of typing `quit`) or the server closes the connection (i.e., when the server is shut down). Each connection to the MySQL server is assigned an identifier, which is shown to you when you first log in:

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 11
Server version: 8.0.15 MySQL Community Server - GPL

Copyright (c) 2000, 2019, 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 buffer.
```

In this case, my connection ID is `11`. This information might be useful to your database administrator if something goes awry, such as a malformed query that runs for hours, so you might want to jot it down.

Once the server has verified your username and password and issued you a connection, you are ready to execute queries (along with other SQL statements). Each time a query is sent to the server, the server checks the following things prior to statement execution:

- Do you have permission to execute the statement?
- Do you have permission to access the desired data?
- Is your statement syntax correct?

If your statement passes these three tests, then your query is handed to the *query optimizer*, whose job it is to determine the most efficient way to execute your query. The optimizer looks at such things as the order in which to join the tables named in your `from` clause and what indexes are available, and then it picks an *execution plan*, which the server uses to execute your query.

NOTE

Understanding and influencing how your database server chooses execution plans is a fascinating topic that many of you will want to explore. For those readers using MySQL, you might consider reading Baron Schwartz et al.'s *High Performance MySQL* (O'Reilly). Among other things, you will learn how to generate indexes, analyze execution plans, influence the optimizer via query hints, and tune your server's startup parameters. If you are using Oracle Database or SQL Server, dozens of tuning books are available.

Once the server has finished executing your query, the *result set* is returned to the calling application (which is, once again, the `mysql` tool). As I mentioned in Chapter 1, a result set is just another table containing rows and columns. If your query fails to yield any results, the `mysql` tool will show you the message found at the end of the following example:

```
mysql> SELECT first_name, last_name
    -> FROM customer
    -> WHERE last_name = 'ZIEGLER';
Empty set (0.02 sec)
```

If the query returns one or more rows, the `mysql` tool will format the results by adding column headers and constructing boxes around the columns using the -, |, and + symbols, as shown in the next example:

```
mysql> SELECT *
    -> FROM category;
+-------------+-------------+---------------------+
```

```
| category_id | name        | last_update         |
+-------------+-------------+---------------------+
|           1 | Action      | 2006-02-15 04:46:27 |
|           2 | Animation   | 2006-02-15 04:46:27 |
|           3 | Children    | 2006-02-15 04:46:27 |
|           4 | Classics    | 2006-02-15 04:46:27 |
|           5 | Comedy      | 2006-02-15 04:46:27 |
|           6 | Documentary | 2006-02-15 04:46:27 |
|           7 | Drama       | 2006-02-15 04:46:27 |
|           8 | Family      | 2006-02-15 04:46:27 |
|           9 | Foreign     | 2006-02-15 04:46:27 |
|          10 | Games       | 2006-02-15 04:46:27 |
|          11 | Horror      | 2006-02-15 04:46:27 |
|          12 | Music       | 2006-02-15 04:46:27 |
|          13 | New         | 2006-02-15 04:46:27 |
|          14 | Sci-Fi      | 2006-02-15 04:46:27 |
|          15 | Sports      | 2006-02-15 04:46:27 |
|          16 | Travel      | 2006-02-15 04:46:27 |
+-------------+-------------+---------------------+
16 rows in set (0.02 sec)
```

This query returns all three columns for of all the rows in the `category` table. After the last row of data is displayed, the `mysql` tool displays a message telling you how many rows were returned, which, in this case, is 16.

Query Clauses

Several components or *clauses* make up the `select` statement. While only one of them is mandatory when using MySQL (the `select` clause), you will usually include at least two or three of the six available clauses. Table 3-1 shows the different clauses and their purposes.

Table 3-1. Query clauses

Clause name	Purpose
`select`	Determines which columns to include in the query's result set
`from`	Identifies the tables from which to retrieve data and how the tables should be joined
`where`	Filters out unwanted data
`group by`	Used to group rows together by common column values
`having`	Filters out unwanted groups
`order by`	Sorts the rows of the final result set by one or more columns

All of the clauses shown in Table 3-1 are included in the ANSI specification. The following sections delve into the uses of the six major query clauses.

The select Clause

Even though the `select` clause is the first clause of a `select` statement, it is one of the last clauses that the database server evaluates. The reason for this is that before you can determine what to include in the final result set, you need to know all of the possible columns that *could* be included in the final result set. In order to fully understand the role of the `select` clause, therefore, you will need to understand a bit about the `from` clause. Here's a query to get started:

```
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 (0.03 sec)
```

In this query, the `from` clause lists a single table (`language`), and the `select` clause indicates that *all* columns (designated by `*`) in the `language` table should be included in the result set. This query could be described in English as follows:

> *Show me all the columns and all the rows in the `language` table.*

In addition to specifying all the columns via the asterisk character, you can explicitly name the columns you are interested in, such as:

```
mysql> SELECT language_id, name, last_update
```

```
    -> 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 results are identical to the first query, since all the columns in the `language` table (`language_id`, `name`, and `last_update`) are named in the `select` clause. You can choose to include only a subset of the columns in the `language` table as well:

```
mysql> SELECT name
    -> FROM language;
+----------+
| name     |
+----------+
| English  |
| Italian  |
| Japanese |
| Mandarin |
| French   |
| German   |
+----------+
6 rows in set (0.00 sec)
```

The job of the `select` clause, therefore, is as follows:

> *The `select` clause determines which of all possible columns should be included in the query's result set.*

If you were limited to including only columns from the table or tables named in the `from` clause, things would be rather dull. However, you can spice things up in your `select` clause by including things such as:

- Literals, such as numbers or strings
- Expressions, such as `transaction.amount * -1`
- Built-in function calls, such as `ROUND(transaction.amount, 2)`
- User-defined function calls

The next query demonstrates the use of a table column, a literal, an expression, and a built-in function call in a single query against the `language` table:

```
mysql> SELECT language_id,
    ->   'COMMON' language_usage,
    ->   language_id * 3.1415927 lang_pi_value,
    ->   upper(name) language_name
    -> FROM language;
+-------------+----------------+---------------+---------------+
| language_id | language_usage | lang_pi_value | language_name |
+-------------+----------------+---------------+---------------+
|           1 | COMMON         |     3.1415927 | ENGLISH       |
|           2 | COMMON         |     6.2831854 | ITALIAN       |
|           3 | COMMON         |     9.4247781 | JAPANESE      |
|           4 | COMMON         |    12.5663708 | MANDARIN      |
|           5 | COMMON         |    15.7079635 | FRENCH        |
|           6 | COMMON         |    18.8495562 | GERMAN        |
+-------------+----------------+---------------+---------------+
6 rows in set (0.04 sec)
```

We cover expressions and built-in functions in detail later, but I wanted to give you a feel for what kinds of things can be included in the `select` clause. If you only need to execute a built-in function or evaluate a simple expression, you can skip the `from` clause entirely. Here's an example:

```
mysql> SELECT version(),
    ->   user(),
    ->   database();
+-----------+----------------+------------+
| version() | user()         | database() |
+-----------+----------------+------------+
| 8.0.15    | root@localhost | sakila     |
+-----------+----------------+------------+
1 row in set (0.00 sec)
```

Since this query simply calls three built-in functions and doesn't retrieve data from any tables, there is no need for a `from` clause.

Column Aliases

Although the `mysql` tool will generate labels for the columns returned by your queries, you may want to assign your own labels. While you might want to assign

a new label to a column from a table (if it is poorly or ambiguously named), you will almost certainly want to assign your own labels to those columns in your result set that are generated by expressions or built-in function calls. You can do so by adding a *column alias* after each element of your `select` clause. Here's the previous query against the `language` table, which included column aliases for three of the columns:

```
mysql> SELECT language_id,
    ->   'COMMON' language_usage,
    ->   language_id * 3.1415927 lang_pi_value,
    ->   upper(name) language_name
    -> FROM language;
+-------------+----------------+---------------+---------------+
| language_id | language_usage | lang_pi_value | language_name |
+-------------+----------------+---------------+---------------+
|           1 | COMMON         |     3.1415927 | ENGLISH       |
|           2 | COMMON         |     6.2831854 | ITALIAN       |
|           3 | COMMON         |     9.4247781 | JAPANESE      |
|           4 | COMMON         |    12.5663708 | MANDARIN      |
|           5 | COMMON         |    15.7079635 | FRENCH        |
|           6 | COMMON         |    18.8495562 | GERMAN        |
+-------------+----------------+---------------+---------------+
6 rows in set (0.04 sec)
```

If you look at the `select` clause, you can see how the column aliases `language_usage`, `lang_pi_value`, and `language_name` are added after the second, third, and fourth columns. I think you will agree that the output is easier to understand with column aliases in place, and it would be easier to work with programmatically if you were issuing the query from within Java or Python rather than interactively via the `mysql` tool. In order to make your column aliases stand out even more, you also have the option of using the `as` keyword before the alias name, as in:

```
mysql> SELECT language_id,
    ->   'COMMON' AS language_usage,
    ->   language_id * 3.1415927 AS lang_pi_value,
    ->   upper(name) AS language_name
    -> FROM language;
```

Many people feel that including the optional `as` keyword improves readability, although I have chosen not to use it for the examples in this book.

Removing Duplicates

In some cases, a query might return duplicate rows of data. For example, if you were to retrieve the IDs of all actors who appeared in a film, you would see the following:

```
mysql> SELECT actor_id FROM film_actor ORDER BY actor_id;
+----------+
| actor_id |
+----------+
|        1 |
|        1 |
|        1 |
|        1 |
|        1 |
|        1 |
|        1 |
|        1 |
|        1 |
|        1 |
...
|      200 |
|      200 |
|      200 |
|      200 |
|      200 |
|      200 |
|      200 |
|      200 |
|      200 |
+----------+
5462 rows in set (0.01 sec)
```

Since some actors appeared in more than one film, you will see the same actor ID multiple times. What you probably want in this case is the *distinct* set of actors, instead of seeing the actor IDs repeated for each film in which they appeared. You can achieve this by adding the keyword `distinct` directly after the `select` keyword, as demonstrated by the following:

```
mysql> SELECT DISTINCT actor_id FROM film_actor ORDER BY actor_id;
+----------+
| actor_id |
+----------+
|        1 |
|        2 |
|        3 |
|        4 |
```

```
|       5 |
|       6 |
|       7 |
|       8 |
|       9 |
|      10 |
...
|     192 |
|     193 |
|     194 |
|     195 |
|     196 |
|     197 |
|     198 |
|     199 |
|     200 |
+----------+
200 rows in set (0.01 sec)
```

The result set now contains 200 rows, one for each distinct actor, rather than 5,462 rows, one for each film appearance by an actor.

NOTE

If you simply want a list of all actors, you can query the `actor` table rather than reading through all the rows in `film_actor` and removing duplicates.

If you do not want the server to remove duplicate data or you are sure there will be no duplicates in your result set, you can specify the `all` keyword instead of specifying `distinct`. However, the `all` keyword is the default and never needs to be explicitly named, so most programmers do not include `all` in their queries.

WARNING

Keep in mind that generating a distinct set of results requires the data to be sorted, which can be time consuming for large result sets. Don't fall into the trap of using `distinct` just to be sure there are no duplicates; instead, take the time to understand the data you are working with so that you will know whether duplicates are possible.

The from Clause

Thus far, you have seen queries whose `from` clauses contain a single table. Although most SQL books define the `from` clause as simply a list of one or more tables, I would like to broaden the definition as follows:

> *The `from` clause defines the tables used by a query, along with the means of linking the tables together.*

This definition is composed of two separate but related concepts, which we explore in the following sections.

Tables

When confronted with the term *table*, most people think of a set of related rows stored in a database. While this does describe one type of table, I would like to use the word in a more general way by removing any notion of how the data might be stored and concentrating on just the set of related rows. Four different types of tables meet this relaxed definition:

- Permanent tables (i.e., created using the `create table` statement)
- Derived tables (i.e., rows returned by a subquery and held in memory)
- Temporary tables (i.e., volatile data held in memory)
- Virtual tables (i.e., created using the `create view` statement)

Each of these table types may be included in a query's `from` clause. By now, you should be comfortable with including a permanent table in a `from` clause, so I will briefly describe the other types of tables that can be referenced in a `from` clause.

Derived (subquery-generated) tables

A subquery is a query contained within another query. Subqueries are surrounded by parentheses and can be found in various parts of a `select` statement; within the `from` clause, however, a subquery serves the role of generating a derived table that is visible from all other query clauses and can interact with other tables named in the `from` clause. Here's a simple example:

```
mysql> SELECT concat(cust.last_name, ', ', cust.first_name) full_name
    -> FROM
    ->  (SELECT first_name, last_name, email
    ->   FROM customer
    ->   WHERE first_name = 'JESSIE'
    ->  ) cust;
+---------------+
| full_name     |
+---------------+
| BANKS, JESSIE |
| MILAM, JESSIE |
```

```
+---------------+
2 rows in set (0.00 sec)
```

In this example, a subquery against the `customer` table returns three columns, and the *containing query* references two of the three available columns. The subquery is referenced by the containing query via its alias, which, in this case, is `cust`. The data in `cust` is held in memory for the duration of the query and is then discarded. This is a simplistic and not particularly useful example of a subquery in a `from` clause; you will find detailed coverage of subqueries in Chapter 9.

Temporary tables

Although the implementations differ, every relational database allows the ability to define volatile, or temporary, tables. These tables look just like permanent tables, but any data inserted into a temporary table will disappear at some point (generally at the end of a transaction or when your database session is closed). Here's a simple example showing how actors whose last names start with J can be stored temporarily:

```
mysql> CREATE TEMPORARY TABLE actors_j
    ->  (actor_id smallint(5),
    ->   first_name varchar(45),
    ->   last_name varchar(45)
    ->  );
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO actors_j
    -> SELECT actor_id, first_name, last_name
    -> FROM actor
    -> WHERE last_name LIKE 'J%';
Query OK, 7 rows affected (0.03 sec)
Records: 7  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM actors_j;
+----------+------------+-----------+
| actor_id | first_name | last_name |
+----------+------------+-----------+
|      119 | WARREN     | JACKMAN   |
|      131 | JANE       | JACKMAN   |
|        8 | MATTHEW    | JOHANSSON |
|       64 | RAY        | JOHANSSON |
|      146 | ALBERT     | JOHANSSON |
|       82 | WOODY      | JOLIE     |
|       43 | KIRK       | JOVOVICH  |
+----------+------------+-----------+
7 rows in set (0.00 sec)
```

These seven rows are held in memory temporarily and will disappear after your session is closed.

NOTE

Most database servers also drop the temporary table when the session ends. The exception is Oracle Database, which keeps the definition of the temporary table available for future sessions.

Views

A view is a query that is stored in the data dictionary. It looks and acts like a table, but there is no data associated with a view (this is why I call it a *virtual* table). When you issue a query against a view, your query is merged with the view definition to create a final query to be executed.

To demonstrate, here's a view definition that queries the `employee` table and includes four of the available columns:

```
mysql> CREATE VIEW cust_vw AS
    -> SELECT customer_id, first_name, last_name, active
    -> FROM customer;
Query OK, 0 rows affected (0.12 sec)
```

When the view is created, no additional data is generated or stored: the server simply tucks away the `select` statement for future use. Now that the view exists, you can issue queries against it, as in:

```
mysql> SELECT first_name, last_name
    -> FROM cust_vw
    -> WHERE active = 0;
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| SANDRA     | MARTIN    |
| JUDITH     | COX       |
| SHEILA     | WELLS     |
| ERICA      | MATTHEWS  |
```

```
| HEIDI      | LARSON    |
| PENNY      | NEAL      |
| KENNETH    | GOODEN    |
| HARRY      | ARCE      |
| NATHAN     | RUNYON    |
| THEODORE   | CULP      |
| MAURICE    | CRAWLEY   |
| BEN        | EASTER    |
| CHRISTIAN  | JUNG      |
| JIMMIE     | EGGLESTON |
| TERRANCE   | ROUSH     |
+------------+-----------+
15 rows in set (0.00 sec)
```

Views are created for various reasons, including to hide columns from users and to simplify complex database designs.

Table Links

The second deviation from the simple `from` clause definition is the mandate that if more than one table appears in the `from` clause, the conditions used to *link* the tables must be included as well. This is not a requirement of MySQL or any other database server, but it is the ANSI-approved method of joining multiple tables, and it is the most portable across the various database servers. We explore joining multiple tables in depth in Chapters 5 and 10, but here's a simple example in case I have piqued your curiosity:

```
mysql> SELECT customer.first_name, customer.last_name,
    ->   time(rental.rental_date) rental_time
    -> FROM customer
    ->   INNER JOIN rental
    ->   ON customer.customer_id = rental.customer_id
    -> WHERE date(rental.rental_date) = '2005-06-14';
+------------+-----------+-------------+
| first_name | last_name | rental_time |
+------------+-----------+-------------+
| JEFFERY    | PINSON    | 22:53:33    |
| ELMER      | NOE       | 22:55:13    |
| MINNIE     | ROMERO    | 23:00:34    |
| MIRIAM     | MCKINNEY  | 23:07:08    |
| DANIEL     | CABRAL    | 23:09:38    |
| TERRANCE   | ROUSH     | 23:12:46    |
| JOYCE      | EDWARDS   | 23:16:26    |
| GWENDOLYN  | MAY       | 23:16:27    |
| CATHERINE  | CAMPBELL  | 23:17:03    |
| MATTHEW    | MAHAN     | 23:25:58    |
| HERMAN     | DEVORE    | 23:35:09    |
| AMBER      | DIXON     | 23:42:56    |
| TERRENCE   | GUNDERSON | 23:47:35    |
| SONIA      | GREGORY   | 23:50:11    |
| CHARLES    | KOWALSKI  | 23:54:34    |
| JEANETTE   | GREENE    | 23:54:46    |
+------------+-----------+-------------+
16 rows in set (0.01 sec)
```

The previous query displays data from both the `customer` table (`first_name`, `last_name`) and the `rental` table (`rental_date`), so both tables are included in the `from` clause. The mechanism for linking the two tables (referred to as a *join*) is the customer ID stored in both the `customer` and `rental` tables. Thus, the database server is instructed to use the value of the `customer_id` column in the `customer` table to find all of the customer's rentals in the `rental` table. Join conditions for the two tables are found in the `on` subclause of the `from` clause; in this case, the join condition is `ON customer.customer_id = rental.customer_id`. The `where` clause is not part of the join and is only included to keep the result set fairly small, since there are more than 16,000 rows in the `rental` table. Again, please refer to Chapter 5 for a thorough discussion of joining multiple tables.

Defining Table Aliases

When multiple tables are joined in a single query, you need a way to identify which table you are referring to when you reference columns in the `select`, `where`, `group by`, `having`, and `order by` clauses. You have two choices when referencing a table outside the `from` clause:

- Use the entire table name, such as `employee.emp_id`.
- Assign each table an *alias* and use the alias throughout the query.

In the previous query, I chose to use the entire table name in the `select` and `on` clauses. Here's what the same query looks like using table aliases:

```
SELECT c.first_name, c.last_name,
  time(r.rental_date) rental_time
FROM customer c
  INNER JOIN rental r
  ON c.customer_id = r.customer_id
WHERE date(r.rental_date) = '2005-06-14';
```

If you look closely at the `from` clause, you will see that the `customer` table is assigned the alias `c`, and the `rental` table is assigned the alias `r`. These aliases are

then used in the `on` clause when defining the join condition as well as in the `select` clause when specifying the columns to include in the result set. I hope you will agree that using aliases makes for a more compact statement without causing confusion (as long as your choices for alias names are reasonable). Additionally, you may use the `as` keyword with your table aliases, similar to what was demonstrated earlier for column aliases:

```
SELECT c.first_name, c.last_name,
  time(r.rental_date) rental_time
FROM customer AS c
  INNER JOIN rental AS r
  ON c.customer_id = r.customer_id
WHERE date(r.rental_date) = '2005-06-14';
```

I have found that roughly half of the database developers I have worked with use the `as` keyword with their column and table aliases, and half do not.

The where Clause

In some cases, you may want to retrieve all rows from a table, especially for small tables such as `language`. Most of the time, however, you will not want to retrieve *every* row from a table but will want a way to filter out those rows that are not of interest. This is a job for the `where` clause.

> *The `where` clause is the mechanism for filtering out unwanted rows from your result set.*

For example, perhaps you are interested in renting a film but you are only interested in movies rated G that can be kept for at least a week. The following query employs a `where` clause to retrieve *only* the films meeting these criteria:

```
mysql> SELECT title
    -> FROM film
    -> WHERE rating = 'G' AND rental_duration >= 7;
+--------------------------+
| title                    |
+--------------------------+
| BLANKET BEVERLY          |
| BORROWERS BEDAZZLED      |
| BRIDE INTRIGUE           |
| CATCH AMISTAD            |
| CITIZEN SHREK            |
| COLDBLOODED DARLING      |
| CONTROL ANTHEM           |
| CRUELTY UNFORGIVEN       |
| DARN FORRESTER           |
| DESPERATE TRAINSPOTTING  |
| DIARY PANIC              |
| DRACULA CRYSTAL          |
| EMPIRE MALKOVICH         |
| FIREHOUSE VIETNAM        |
| GILBERT PELICAN          |
| GRADUATE LORD            |
| GREASE YOUTH             |
| GUN BONNIE               |
| HOOK CHARIOTS            |
| MARRIED GO               |
| MENAGERIE RUSHMORE       |
| MUSCLE BRIGHT            |
| OPERATION OPERATION      |
| PRIMARY GLASS            |
| REBEL AIRPORT            |
| SPIKING ELEMENT          |
| TRUMAN CRAZY             |
| WAKE JAWS                |
| WAR NOTTING              |
+--------------------------+
29 rows in set (0.00 sec)
```

In this case, the `where` clause filtered out 971 of the 1000 rows in the `film` table. This `where` clause contains two *filter conditions*, but you can include as many conditions as are required; individual conditions are separated using operators such as `and`, `or`, and `not` (see Chapter 4 for a complete discussion of the `where` clause and filter conditions).

Let's see what would happen if you change the operator separating the two conditions from `and` to `or`:

```
mysql> SELECT title
    -> FROM film
    -> WHERE rating = 'G' OR rental_duration >= 7;
+----------------------------+
| title                      |
+----------------------------+
| ACE GOLDFINGER             |
| ADAPTATION HOLES           |
| AFFAIR PREJUDICE           |
| AFRICAN EGG                |
| ALAMO VIDEOTAPE            |
| AMISTAD MIDSUMMER          |
| ANGELS LIFE                |
```

```
| ANNIE IDENTITY           |
|...                       |
| WATERSHIP FRONTIER       |
| WEREWOLF LOLA            |
| WEST LION                |
| WESTWARD SEABISCUIT      |
| WOLVES DESIRE            |
| WON DARES                |
| WORKER TARZAN            |
| YOUNG LANGUAGE           |
+--------------------------+
340 rows in set (0.00 sec)
```

When you separate conditions using the `and` operator, *all* conditions must evaluate to `true` to be included in the result set; when you use `or`, however, only *one* of the conditions needs to evaluate to `true` for a row to be included, which explains why the size of the result set has jumped from 29 to 340 rows.

So, what should you do if you need to use both `and` and `or` operators in your `where` clause? Glad you asked. You should use parentheses to group conditions together. The next query specifies that only those films that are rated G and are available for 7 or more days, or are rated PG-13 and are available 3 or fewer days, be included in the result set:

```
mysql> SELECT title, rating, rental_duration
    -> FROM film
    -> WHERE (rating = 'G' AND rental_duration >= 7)
    ->   OR (rating = 'PG-13' AND rental_duration < 4);
+-------------------------+--------+-----------------+
| title                   | rating | rental_duration |
+-------------------------+--------+-----------------+
| ALABAMA DEVIL           | PG-13  |               3 |
| BACKLASH UNDEFEATED     | PG-13  |               3 |
| BILKO ANONYMOUS         | PG-13  |               3 |
| BLANKET BEVERLY         | G      |               7 |
| BORROWERS BEDAZZLED     | G      |               7 |
| BRIDE INTRIGUE          | G      |               7 |
| CASPER DRAGONFLY        | PG-13  |               3 |
| CATCH AMISTAD           | G      |               7 |
| CITIZEN SHREK           | G      |               7 |
| COLDBLOODED DARLING     | G      |               7 |
|...                                                 |
| TREASURE COMMAND        | PG-13  |               3 |
| TRUMAN CRAZY            | G      |               7 |
| WAIT CIDER              | PG-13  |               3 |
| WAKE JAWS               | G      |               7 |
| WAR NOTTING             | G      |               7 |
| WORLD LEATHERNECKS      | PG-13  |               3 |
+-------------------------+--------+-----------------+
68 rows in set (0.00 sec)
```

You should always use parentheses to separate groups of conditions when mixing different operators so that you, the database server, and anyone who comes along later to modify your code will be on the same page.

The group by and having Clauses

All the queries thus far have retrieved raw data without any manipulation. Sometimes, however, you will want to find trends in your data that will require the database server to cook the data a bit before you retrieve your result set. One such mechanism is the `group by` clause, which is used to group data by column values. For example, let's say you wanted to find all of the customers who have rented 40 or more films. Rather than looking through all 16,044 rows in the `rental` table, you can write a query that instructs the server to group all rentals by customer, count the number of rentals for each customer, and then return only those customers whose rental count is at least 40. When using the `group by` clause to generate groups of rows, you may also use the `having` clause, which allows you to filter grouped data in the same way the `where` clause lets you filter raw data.

Here's what the query looks like:

```
mysql> SELECT c.first_name, c.last_name, count(*)
    -> FROM customer c
    ->   INNER JOIN rental r
    ->   ON c.customer_id = r.customer_id
    -> GROUP BY c.first_name, c.last_name
    -> HAVING count(*) >= 40;
+------------+-----------+----------+
| first_name | last_name | count(*) |
+------------+-----------+----------+
| TAMMY      | SANDERS   |       41 |
| CLARA      | SHAW      |       42 |
| ELEANOR    | HUNT      |       46 |
| SUE        | PETERS    |       40 |
| MARCIA     | DEAN      |       42 |
| WESLEY     | BULL      |       40 |
| KARL       | SEAL      |       45 |
```

```
+------------+-----------+----------+
7 rows in set (0.03 sec)
```

I wanted to briefly mention these two clauses so that they don't catch you by surprise later in the book, but they are a bit more advanced than the other four `select` clauses. Therefore, I ask that you wait until Chapter 8 for a full description of how and when to use `group by` and `having`.

The order by Clause

In general, the rows in a result set returned from a query are not in any particular order. If you want your result set to be sorted, you will need to instruct the server to sort the results using the `order by` clause:

> *The `order by` clause is the mechanism for sorting your result set using either raw column data or expressions based on column data.*

For example, here's another look at an earlier query that returns all customers who rented a film on June 14, 2005:

```
mysql> SELECT c.first_name, c.last_name,
    ->   time(r.rental_date) rental_time
    -> FROM customer c
    ->   INNER JOIN rental r
    ->   ON c.customer_id = r.customer_id
    -> WHERE date(r.rental_date) = '2005-06-14';
+------------+-----------+-------------+
| first_name | last_name | rental_time |
+------------+-----------+-------------+
| JEFFERY    | PINSON    | 22:53:33    |
| ELMER      | NOE       | 22:55:13    |
| MINNIE     | ROMERO    | 23:00:34    |
| MIRIAM     | MCKINNEY  | 23:07:08    |
| DANIEL     | CABRAL    | 23:09:38    |
| TERRANCE   | ROUSH     | 23:12:46    |
| JOYCE      | EDWARDS   | 23:16:26    |
| GWENDOLYN  | MAY       | 23:16:27    |
| CATHERINE  | CAMPBELL  | 23:17:03    |
| MATTHEW    | MAHAN     | 23:25:58    |
| HERMAN     | DEVORE    | 23:35:09    |
| AMBER      | DIXON     | 23:42:56    |
| TERRENCE   | GUNDERSON | 23:47:35    |
| SONIA      | GREGORY   | 23:50:11    |
| CHARLES    | KOWALSKI  | 23:54:34    |
| JEANETTE   | GREENE    | 23:54:46    |
+------------+-----------+-------------+
16 rows in set (0.01 sec)
```

If you would like the results to be in alphabetical order by last name, you can add the `last_name` column to the `order by` clause:

```
mysql> SELECT c.first_name, c.last_name,
    ->   time(r.rental_date) rental_time
    -> FROM customer c
    ->   INNER JOIN rental r
    ->   ON c.customer_id = r.customer_id
    -> WHERE date(r.rental_date) = '2005-06-14'
    -> ORDER BY c.last_name;
+------------+-----------+-------------+
| first_name | last_name | rental_time |
+------------+-----------+-------------+
| DANIEL     | CABRAL    | 23:09:38    |
| CATHERINE  | CAMPBELL  | 23:17:03    |
| HERMAN     | DEVORE    | 23:35:09    |
| AMBER      | DIXON     | 23:42:56    |
| JOYCE      | EDWARDS   | 23:16:26    |
| JEANETTE   | GREENE    | 23:54:46    |
| SONIA      | GREGORY   | 23:50:11    |
| TERRENCE   | GUNDERSON | 23:47:35    |
| CHARLES    | KOWALSKI  | 23:54:34    |
| MATTHEW    | MAHAN     | 23:25:58    |
| GWENDOLYN  | MAY       | 23:16:27    |
| MIRIAM     | MCKINNEY  | 23:07:08    |
| ELMER      | NOE       | 22:55:13    |
| JEFFERY    | PINSON    | 22:53:33    |
| MINNIE     | ROMERO    | 23:00:34    |
| TERRANCE   | ROUSH     | 23:12:46    |
+------------+-----------+-------------+
16 rows in set (0.01 sec)
```

While it is not the case in this example, large customer lists will often contain multiple people having the same last name, so you may want to extend the sort criteria to include the person's first name as well.

You can accomplish this by adding the `first_name` column after the `last_name` column in the `order by` clause:

```
mysql> SELECT c.first_name, c.last_name,
    ->   time(r.rental_date) rental_time
    -> FROM customer c
    ->   INNER JOIN rental r
```

```
    ->   ON c.customer_id = r.customer_id
    -> WHERE date(r.rental_date) = '2005-06-14'
    -> ORDER BY c.last_name, c.first_name;
+------------+-----------+-------------+
| first_name | last_name | rental_time |
+------------+-----------+-------------+
| DANIEL     | CABRAL    | 23:09:38    |
| CATHERINE  | CAMPBELL  | 23:17:03    |
| HERMAN     | DEVORE    | 23:35:09    |
| AMBER      | DIXON     | 23:42:56    |
| JOYCE      | EDWARDS   | 23:16:26    |
| JEANETTE   | GREENE    | 23:54:46    |
| SONIA      | GREGORY   | 23:50:11    |
| TERRENCE   | GUNDERSON | 23:47:35    |
| CHARLES    | KOWALSKI  | 23:54:34    |
| MATTHEW    | MAHAN     | 23:25:58    |
| GWENDOLYN  | MAY       | 23:16:27    |
| MIRIAM     | MCKINNEY  | 23:07:08    |
| ELMER      | NOE       | 22:55:13    |
| JEFFERY    | PINSON    | 22:53:33    |
| MINNIE     | ROMERO    | 23:00:34    |
| TERRANCE   | ROUSH     | 23:12:46    |
+------------+-----------+-------------+
16 rows in set (0.01 sec)
```

The order in which columns appear in your `order by` clause does make a difference when you include more than one column. If you were to switch the order of the two columns in the `order by` clause, Amber Dixon would appear first in the result set.

Ascending Versus Descending Sort Order

When sorting, you have the option of specifying *ascending* or *descending* order via the `asc` and `desc` keywords. The default is ascending, so you will need to add the `desc` keyword if you want to use a descending sort. For example, the following query shows all customers who rented films on June 14, 2005, in descending order of rental time:

```
mysql> SELECT c.first_name, c.last_name,
    ->   time(r.rental_date) rental_time
    -> FROM customer c
    ->   INNER JOIN rental r
    ->   ON c.customer_id = r.customer_id
    -> WHERE date(r.rental_date) = '2005-06-14'
    -> ORDER BY time(r.rental_date) desc;
+------------+-----------+-------------+
| first_name | last_name | rental_time |
+------------+-----------+-------------+
| JEANETTE   | GREENE    | 23:54:46    |
| CHARLES    | KOWALSKI  | 23:54:34    |
| SONIA      | GREGORY   | 23:50:11    |
| TERRENCE   | GUNDERSON | 23:47:35    |
| AMBER      | DIXON     | 23:42:56    |
| HERMAN     | DEVORE    | 23:35:09    |
| MATTHEW    | MAHAN     | 23:25:58    |
| CATHERINE  | CAMPBELL  | 23:17:03    |
| GWENDOLYN  | MAY       | 23:16:27    |
| JOYCE      | EDWARDS   | 23:16:26    |
| TERRANCE   | ROUSH     | 23:12:46    |
| DANIEL     | CABRAL    | 23:09:38    |
| MIRIAM     | MCKINNEY  | 23:07:08    |
| MINNIE     | ROMERO    | 23:00:34    |
| ELMER      | NOE       | 22:55:13    |
| JEFFERY    | PINSON    | 22:53:33    |
+------------+-----------+-------------+
16 rows in set (0.01 sec)
```

Descending sorts are commonly used for ranking queries, such as “show me the top five account balances.” MySQL includes a `limit` clause that allows you to sort your data and then discard all but the first *X* rows.

Sorting via Numeric Placeholders

If you are sorting using the columns in your `select` clause, you can opt to reference the columns by their *position* in the `select` clause rather than by name. This can be especially helpful if you are sorting on an expression, such as in the previous example. Here’s the previous example one last time, with an `order by` clause specifying a descending sort using the third element in the `select` clause:

```
mysql> SELECT c.first_name, c.last_name,
    ->   time(r.rental_date) rental_time
    -> FROM customer c
    ->   INNER JOIN rental r
    ->   ON c.customer_id = r.customer_id
    -> WHERE date(r.rental_date) = '2005-06-14'
    -> ORDER BY 3 desc;
+------------+-----------+-------------+
| first_name | last_name | rental_time |
+------------+-----------+-------------+
| JEANETTE   | GREENE    | 23:54:46    |
| CHARLES    | KOWALSKI  | 23:54:34    |
| SONIA      | GREGORY   | 23:50:11    |
| TERRENCE   | GUNDERSON | 23:47:35    |
| AMBER      | DIXON     | 23:42:56    |
```

```
| HERMAN     | DEVORE    | 23:35:09    |
| MATTHEW    | MAHAN     | 23:25:58    |
| CATHERINE  | CAMPBELL  | 23:17:03    |
| GWENDOLYN  | MAY       | 23:16:27    |
| JOYCE      | EDWARDS   | 23:16:26    |
| TERRANCE   | ROUSH     | 23:12:46    |
| DANIEL     | CABRAL    | 23:09:38    |
| MIRIAM     | MCKINNEY  | 23:07:08    |
| MINNIE     | ROMERO    | 23:00:34    |
| ELMER      | NOE       | 22:55:13    |
| JEFFERY    | PINSON    | 22:53:33    |
+------------+-----------+-------------+
16 rows in set (0.01 sec)
```

You might want to use this feature sparingly, since adding a column to the `select` clause without changing the numbers in the `order by` clause can lead to unexpected results. Personally, I may reference columns positionally when writing ad hoc queries, but I always reference columns by name when writing code.

Test Your Knowledge

The following exercises are designed to strengthen your understanding of the `select` statement and its various clauses. Please see Appendix B for solutions.

Exercise 3-1

Retrieve the actor ID, first name, and last name for all actors. Sort by last name and then by first name.

Exercise 3-2

Retrieve the actor ID, first name, and last name for all actors whose last name equals `'WILLIAMS'` or `'DAVIS'`.

Exercise 3-3

Write a query against the `rental` table that returns the IDs of the customers who rented a film on July 5, 2005 (use the `rental.rental_date` column, and you can use the `date()` function to ignore the time component). Include a single row for each distinct customer ID.

Exercise 3-4

Fill in the blanks (denoted by `< # >`) for this multitable query to achieve the following results:

```
mysql> SELECT c.email, r.return_date
    -> FROM customer c
    ->   INNER JOIN rental <1>
    ->   ON c.customer_id = <2>
    -> WHERE date(r.rental_date) = '2005-06-14'
    -> ORDER BY <3> <4>;
+--------------------------------------+---------------------+
| email                                | return_date         |
+--------------------------------------+---------------------+
| DANIEL.CABRAL@sakilacustomer.org     | 2005-06-23 22:00:38 |
| TERRANCE.ROUSH@sakilacustomer.org    | 2005-06-23 21:53:46 |
| MIRIAM.MCKINNEY@sakilacustomer.org   | 2005-06-21 17:12:08 |
| GWENDOLYN.MAY@sakilacustomer.org     | 2005-06-20 02:40:27 |
| JEANETTE.GREENE@sakilacustomer.org   | 2005-06-19 23:26:46 |
| HERMAN.DEVORE@sakilacustomer.org     | 2005-06-19 03:20:09 |
| JEFFERY.PINSON@sakilacustomer.org    | 2005-06-18 21:37:33 |
| MATTHEW.MAHAN@sakilacustomer.org     | 2005-06-18 05:18:58 |
| MINNIE.ROMERO@sakilacustomer.org     | 2005-06-18 01:58:34 |
| SONIA.GREGORY@sakilacustomer.org     | 2005-06-17 21:44:11 |
| TERRENCE.GUNDERSON@sakilacustomer.org | 2005-06-17 05:28:35 |
| ELMER.NOE@sakilacustomer.org         | 2005-06-17 02:11:13 |
| JOYCE.EDWARDS@sakilacustomer.org     | 2005-06-16 21:00:26 |
| AMBER.DIXON@sakilacustomer.org       | 2005-06-16 04:02:56 |
| CHARLES.KOWALSKI@sakilacustomer.org  | 2005-06-16 02:26:34 |
| CATHERINE.CAMPBELL@sakilacustomer.org | 2005-06-15 20:43:03 |
+--------------------------------------+---------------------+
16 rows in set (0.03 sec)
```