Chapter 5. Advanced Querying

Over the previous two chapters, you've completed an introduction to the basic features of querying and modifying databases with SQL. You should now be able to create, modify, and remove database structures, as well as work with data as you read, insert, delete, and update entries. Over this and the next two chapters, we'll look at more advanced concepts and then will proceed to more administrative and operations-oriented content. You can skim these chapters and return to read them thoroughly when you're comfortable using MySQL.

This chapter teaches you more about querying, giving you skills to answer complex information needs. You'll learn how to do the following:

- Use nicknames, or *aliases*, in queries to save typing and allow a table to be used more than once in a query.
- Aggregate data into groups so you can discover sums, averages, and counts.
- Join tables in different ways.
- Use nested queries.
- Save query results in variables so they can be reused in other queries.

Aliases

Aliases are nicknames. They give you a shorthand way of expressing a column, table, or function name, allowing you to:

- Write shorter queries.
- Express your queries more clearly.
- Use one table in two or more ways in a single query.
- Access data more easily from programs.
- Use special types of nested queries, discussed in "Nested Queries".

Column Aliases

Column aliases are useful for improving the expression of your queries, reducing the number of characters you need to type, and making it easier to work with programming languages such as Python or PHP. Consider a simple, not-very-useful example:

```
mysql> SELECT first_name AS 'First Name', last_name AS
    -> FROM actor LIMIT 5;
```

```
+------------+--------------+
| First Name | Last Name    |
+------------+--------------+
| PENELOPE   | GUINESS      |
| NICK       | WAHLBERG     |
| ED         | CHASE        |
| JENNIFER   | DAVIS        |
| JOHNNY     | LOLLOBRIGIDA |
+------------+--------------+
5 rows in set (0.00 sec)
```

The column `first_name` is aliased as `First Name`, and column `last_name` as `Last Name`. You can see that in the output, the usual column headings, `first_name` and `last_name`, are replaced by the aliases `First Name` and `Last Name`. The advantage is that the aliases might be more meaningful to users. In this case, at the very least, they are more human-readable. Other than that, it's not very useful, but it does illustrate the idea that for a column, you add the keyword `AS` and then a string that represents what you'd like the column to be known as. Specifying the `AS` keyword is not required but makes things much clearer.

NOTE

We'll be using the `LIMIT` clause extensively throughout this chapter, as otherwise almost every output would be unwieldy and long. Sometimes we'll mention that explicitly, sometimes not. You can experiment on your own by removing `LIMIT` from the queries we give. More information about the `LIMIT` clause can be found in "The LIMIT Clause".

Now let's see column aliases doing something useful. Here's an example that uses a MySQL function and an `ORDER BY` clause:

```
mysql> SELECT CONCAT(first_name, ' ', last_name, ' play
    -> FROM actor JOIN film_actor USING (actor_id)
    -> JOIN film USING (film_id)
    -> ORDER BY movie LIMIT 20;
```

```
+--------------------------------------------+
| movie                                      |
+--------------------------------------------+
| ADAM GRANT played in ANNIE IDENTITY        |
| ADAM GRANT played in BALLROOM MOCKINGBIRD  |
| ...                                        |
| ADAM GRANT played in TWISTED PIRATES       |
| ADAM GRANT played in WANDA CHAMBER         |
| ADAM HOPPER played in BLINDNESS GUN        |
| ADAM HOPPER played in BLOOD ARGONAUTS      |
+--------------------------------------------+
20 rows in set (0.03 sec)
```

The MySQL function `CONCAT()` *concatenates* the strings that are parameters—in this case, the `first_name`, a constant string with a space, the `last_name`, the constant string `played in`, and the `title`—to give output such as `ZERO CAGE played in CANYON STOCK`. We've added an alias to the function, `AS movie`, so that we can refer to it easily as `movie` throughout the query. You can see that we do this in the `ORDER BY` clause, where we ask MySQL to sort the output by ascending `movie` value. This is much better than the unaliased alternative, which requires you to write out the `CONCAT()` function again:

```
mysql> SELECT CONCAT(first_name, ' ', last_name, ' play
    -> FROM actor JOIN film_actor USING (actor_id)
    -> JOIN film USING (film_id)
    -> ORDER BY CONCAT(first_name, ' ', last_name, ' pl
    -> LIMIT 20;
```

```
+--------------------------------------------+
| movie                                      |
+-------------------------------------------+
| ADAM GRANT played in ANNIE IDENTITY       |
| ADAM GRANT played in BALLROOM MOCKINGBIRD |
| ...                                       |
| ADAM GRANT played in TWISTED PIRATES      |
| ADAM GRANT played in WANDA CHAMBER        |
| ADAM HOPPER played in BLINDNESS GUN       |
| ADAM HOPPER played in BLOOD ARGONAUTS     |
+-------------------------------------------+
20 rows in set (0.03 sec)
```

The alternative is unwieldy, and worse, you risk mistyping some part of the `ORDER BY` clause and getting a result different from what you expect. (Note that we've used `AS movie` on the first line so that the displayed column has the label `movie`.)

There are restrictions on where you can use column aliases. You can't use them in a `WHERE` clause, or in the `USING` and `ON` clauses that we discuss later in this chapter. This means you can't write a query like this:

```
mysql> SELECT first_name AS name FROM actor WHERE name
```

```
ERROR 1054 (42S22): Unknown column 'name' in 'where cla
```

You can't do that because MySQL doesn't always know the column values before it executes the `WHERE` clause. However, you can use column aliases in the `ORDER BY` clause, and in the `GROUP BY` and `HAVING` clauses discussed later in this chapter.

The `AS` keyword is optional, as we've mentioned. Because of this, the following two queries are equivalent:

```
mysql> SELECT actor_id AS id FROM actor WHERE first_nam
```

```
+----+
| id |
+----+
| 11 |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT actor_id id FROM actor WHERE first_name =
```

```
+-----+
| id |
+-----+
| 11 |
+-----+
1 row in set (0.00 sec)
```

We recommend using the `AS` keyword, since it helps to clearly distinguish an aliased column, especially where you're selecting multiple columns from a list of columns separated by commas.

Alias names have a few restrictions. They can be at most 255 characters in length and can contain any character. Aliases don't always need to be quoted, and they follow the same rules as table and column names do, which we described in Chapter 4. If an alias is a single word and doesn't include special symbols—like a dash, a plus sign, or a space, for example—and is not a keyword, like `USE`, then you don't need to put quotes around it. Otherwise, you need to quote the alias, which you can do using double quotes, single quotes, or backticks. We recommend using lowercase alphanumeric strings for alias names and using a consistent character choice—such as an underscore—to separate words. Aliases are case-insensitive on all platforms.

Table Aliases

Table aliases are useful for the same reasons as column aliases, but they are also sometimes the only way to express a query. This section shows you how to use table aliases, and "Nested Queries" shows you some other sample queries where table aliases are essential.

Here's a basic table alias example that shows you how to save some typing:

```
mysql> SELECT ac.actor_id, ac.first_name, ac.last_name,
    -> actor AS ac INNER JOIN film_actor AS fla USING (
    -> INNER JOIN film AS fl USING (film_id)
    -> WHERE fl.title = 'AFFAIR PREJUDICE';
```

```
+----------+------------+-----------+-----------------
| actor_id | first_name | last_name | title
+----------+------------+-----------+-----------------
|       41 | JODIE      | DEGENERES | AFFAIR PREJUDICE
|       81 | SCARLETT   | DAMON     | AFFAIR PREJUDICE
|       88 | KENNETH    | PESCI     | AFFAIR PREJUDICE
|      147 | FAY        | WINSLET   | AFFAIR PREJUDICE
|      162 | OPRAH      | KILMER    | AFFAIR PREJUDICE
+----------+------------+-----------+-----------------
5 rows in set (0.00 sec)
```

You can see that the `film` and `actor` tables are aliased as `fl` and `ac`, respectively, using the `AS` keyword. This allows you to express column names more compactly, such as `fl.title`. Notice also that you can use table aliases in the `WHERE` clause; unlike column aliases, there are no restrictions on where table aliases can be used in queries. From our example, you can see that we're referring to the table aliases in `SELECT` before they have been defined in `FROM`. There is, however, a catch with table aliases: if an alias has been used for a table, it's impossible to refer to that table without using its new alias. For example, the following statement will error out, as it would if we'd mentioned `film` in the `SELECT` clause:

```
mysql> SELECT ac.actor_id, ac.first_name, ac.last_name,
    -> actor AS ac INNER JOIN film_actor AS fla USING (
    -> INNER JOIN film AS fl USING (film_id)
    -> WHERE film.title = 'AFFAIR PREJUDICE';
```

```
ERROR 1054 (42S22): Unknown column 'film.title' in 'whe
```

As with column aliases, the `AS` keyword is optional. This means that:

```
actor AS ac INNER JOIN film_actor AS fla
```

is the same as

```
actor ac INNER JOIN film_actor fla
```

Again, we prefer the `AS` style because it’s clearer to anyone looking at your queries than the alternative. The length and content restrictions on table aliases names are the same as for column aliases, and our recommendations on choosing them are the same, too.

As discussed in the introduction to this section, table aliases allow you to write queries that you can’t otherwise easily express. Consider an example: suppose you want to know whether two or more films in our collection have the same title, and if so, what those films are. Let’s think about the basic requirement: you want to know if two movies have the same name. To do get that, you might try a query like this:

```
mysql> SELECT * FROM film WHERE title = title;
```

But that doesn’t make sense—every film has the same title as itself, so the query just produces all films as output:

```
+---------+------------------...
| film_id | title            ...
+---------+------------------...
|       1 | ACADEMY DINOSAUR ...
|       2 | ACE GOLDFINGER   ...
|       3 | ADAPTATION HOLES ...
|     ...                    ...
|    1000 | ZORRO ARK        ...
+---------+------------------...
1000 rows in set (0.01 sec)
```

What you really want is to know whether two different films from the `film` table have the same name. But how can you do that in a single query? The answer is to give the table two different aliases; you then check to see whether one row in the first aliased table matches a row in the second:

```
mysql> SELECT m1.film_id, m2.title
    -> FROM film AS m1, film AS m2
    -> WHERE m1.title = m2.title;
```

```
+---------+-------------------+
| film_id | title             |
+---------+-------------------+
|       1 | ACADEMY DINOSAUR  |
|       2 | ACE GOLDFINGER    |
|       3 | ADAPTATION HOLES  |
|     ...                     |
|    1000 | ZORRO ARK         |
+---------+-------------------+
1000 rows in set (0.02 sec)
```

But it still doesn't work! We get all 1,000 movies as answers. The reason is that again, each film matches itself because it occurs in both aliased tables.

To get the query to work, we need to make sure a movie from one aliased table doesn't match itself in the other aliased table. The way to do that is to specify that the movies in each table shouldn't have the same ID:

```
mysql> SELECT m1.film_id, m2.title
    -> FROM film AS m1, film AS m2
    -> WHERE m1.title = m2.title
    -> AND m1.film_id <> m2.film_id;
```

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

You can now see that there aren't two films in the database with the same name. The additional `AND m1.film_id != m2.film_id` stops answers from being reported where the movie ID is the same in both tables.

Table aliases are also useful in nested queries that use the `EXISTS` and `ON` clauses. We'll show you examples later in this chapter when we introduce nested queries.

Aggregating Data

Aggregate functions allow you to discover the properties of a group of rows. You use them for purposes such as finding out how many rows there are in a table, how many rows in a table share a property (such as having the same name or date of birth), finding averages (such as the average temperature in

November), or finding the maximum or minimum values of rows that meet some condition (such as finding the coldest day in August).

This section explains the `GROUP BY` and `HAVING` clauses, the two most commonly used SQL statements for aggregation. But first it explains the `DISTINCT` clause, which is used to report unique results for the output of a query. When neither the `DISTINCT` nor the `GROUP BY` clause is specified, the returned raw data can still be processed using the aggregate functions that we describe in this section.

The DISTINCT Clause

To begin our discussion of aggregate functions, we'll focus on the `DISTINCT` clause. This isn't really an aggregate function, but more of a post-processing filter that allows you to remove duplicates. We've added it into this section because, like aggregate functions, it's concerned with picking examples from the output of a query, rather than processing individual rows.

An example is the best way to understand `DISTINCT`. Consider this query:

```
mysql> SELECT DISTINCT first_name
    -> FROM actor JOIN film_actor USING (actor_id);
```

```
+-------------+
| first_name  |
+-------------+
| PENELOPE    |
| NICK        |
| ...         |
| GREGORY     |
| JOHN        |
| BELA        |
| THORA       |
+-------------+
128 rows in set (0.00 sec)
```

The query finds all first names of all the actors listed in our database that have participated in a film and reports one example of each name. If you remove the `DISTINCT` clause, you get one row of output for each role in every film we have in our database, or 5,462 rows. That's a lot of output, so we're limiting it to five rows, but you can spot the difference immediately with names being repeated:

```
mysql> SELECT first_name
    -> FROM actor JOIN film_actor USING (actor_id)
    -> LIMIT 5;
```

```
+------------+
| first_name |
+------------+
| PENELOPE   |
| PENELOPE   |
| PENELOPE   |
| PENELOPE   |
| PENELOPE   |
+------------+
5 rows in set (0.00 sec)
```

So, the `DISTINCT` clause helps you get a summary.

The `DISTINCT` clause applies to the query output and removes rows that have identical values in the columns selected for output in the query. If you rephrase the previous query to output both `first_name` and `last_name` (but otherwise don't change the `JOIN` clause and still use `DISTINCT`), you'll get 199 rows in the output (that's why we use last names):

```
mysql> SELECT DISTINCT first_name, last_name
    -> FROM actor JOIN film_actor USING (actor_id);
```

```
+-------------+--------------+
| first_name  | last_name    |
+-------------+--------------+
| PENELOPE    | GUINESS      |
| NICK        | WAHLBERG     |
| ...                        |
| JULIA       | FAWCETT      |
| THORA       | TEMPLE       |
+-------------+--------------+
199 rows in set (0.00 sec)
```

Unfortunately, people's names even when last names are added still make for a bad unique key. There are 200 rows in the `actor` table in the `sakila` database, and we're missing one of them. You should remember this issue, as using `DISTINCT` indiscriminately may result in incorrect query results.

To remove duplicates, MySQL needs to sort the output. If indexes are available that are in the same order as required for the sort, or the data itself is in an order that's useful, this process has very little overhead. However, for large tables and without an easy way of accessing the data in the right order, sorting can be very slow. You should use `DISTINCT` (and other aggregate functions) with caution on large datasets. If you do use it, you can check its behavior using the `EXPLAIN` statement discussed in Chapter 7.

The GROUP BY Clause

The `GROUP BY` clause groups output data for the purpose of aggregation. Particularly, that allows us to use aggregate functions (covered in "Aggregate functions") on our data when our projection (that is, the contents of the `SELECT` clause) contains columns other than those within an aggregate function. `GROUP BY` is similar to `ORDER BY` in that it takes a list of columns as an argument. However, these clauses are evaluated at different times and are only similar in how they look, not how they operate.

Let's take a look at a few `GROUP BY` examples that will demonstrate what it can be used for. In its most basic form, when we list every column we `SELECT` in `GROUP BY`, we end up with a `DISTINCT` equivalent. We've already established that a first name is not a unique identifier for an actor:

```
mysql> SELECT first_name FROM actor
    -> WHERE first_name IN ('GENE', 'MERYL');
```

```
+------------+
| first_name |
+------------+
| GENE       |
| GENE       |
| MERYL      |
| GENE       |
| MERYL      |
+------------+
5 rows in set (0.00 sec)
```

We can tell MySQL to group the output by a given column to get rid of duplicates. In this case, we've selected only one column, so let's use that:

```
mysql> SELECT first_name FROM actor
    -> WHERE first_name IN ('GENE', 'MERYL')
    -> GROUP BY first_name;
```

```
+------------+
| first_name |
+------------+
| GENE       |
| MERYL      |
+------------+
2 rows in set (0.00 sec)
```

You can see that the original five rows were folded—or, more accurately, grouped—into just two resulting rows. That's not very helpful, as `DISTINCT` could do the same. It's worth mentioning, however, that this is not always going to be the case. `DISTINCT` and `GROUP BY` are evaluated and executed at different stages of query execution, so you should not confuse them, even if sometimes the effects are similar.

According to the SQL standard, every column projected in the `SELECT` clause that is not part of an aggregate function should be listed in the `GROUP BY` clause. The only time this rule may be violated is when the resulting groups have only one row each. If you think about it, that's logical: if you select `first_name` and `last_name` from the `actor` table and group only by `first_name`, how should the database behave? It cannot output more than one row with the same first name, as that goes against the grouping rules, but there may be more than one last name for a given first name.

For a long time, MySQL extended the standard by allowing you to `GROUP BY` based on fewer columns than defined in `SELECT`. What did it do with the extra columns? Well, it output some value in a nondeterministic way. For example, when you grouped by first name but not by the last name, you could get either of the two rows `GENE, WILLIS` and `GENE, HOPKINS`. That's a nonstandard and dangerous behavior. Imagine that for a year you got `Hopkins`, as if the results were ordered alphabetically, and came to rely on that—but then the table was reorganized, and the order changed. We firmly believe that the SQL standard is correct to limit such behaviors, to avoid unpredictability.

Note also that while every column in the `SELECT` must be used either in `GROUP BY` or in an aggregate function, you can `GROUP BY` columns that are not part of the `SELECT`. You'll see some examples of that later.

Now let's construct a more useful example. An actor usually takes part in many films throughout their career. We may want to find out just how many films a particular actor has played in, or do a calculation for each actor we know of and get a rating by productivity. To start, we can use the techniques we've learned so far and perform an `INNER JOIN` between the `actor` and `film_actor` tables. We don't need the `film` table as we're not looking for any details on the films themselves. We can then order the output by actor's name, making it easier to count what we want:

```
mysql> SELECT first_name, last_name, film_id
    -> FROM actor INNER JOIN film_actor USING (actor_ic
    -> ORDER BY first_name, last_name LIMIT 20;
```

```
+------------+-----------+---------+
| first_name | last_name | film_id |
+------------+-----------+---------+
| ADAM       | GRANT     |      26 |
| ADAM       | GRANT     |      52 |
| ADAM       | GRANT     |     233 |
| ADAM       | GRANT     |     317 |
| ADAM       | GRANT     |     359 |
| ADAM       | GRANT     |     362 |
| ADAM       | GRANT     |     385 |
| ADAM       | GRANT     |     399 |
| ADAM       | GRANT     |     450 |
| ADAM       | GRANT     |     532 |
| ADAM       | GRANT     |     560 |
| ADAM       | GRANT     |     574 |
| ADAM       | GRANT     |     638 |
| ADAM       | GRANT     |     773 |
| ADAM       | GRANT     |     833 |
| ADAM       | GRANT     |     874 |
| ADAM       | GRANT     |     918 |
| ADAM       | GRANT     |     956 |
| ADAM       | HOPPER    |      81 |
| ADAM       | HOPPER    |      82 |
+------------+-----------+---------+
20 rows in set (0.01 sec)
```

By running down the list, it's easy to count off how many films we've got for each actor, or at least for Adam Grant. Without a `LIMIT` , however, the query would return 5,462 distinct rows, and calculating our counts manually would take a lot of time. The `GROUP BY` clause can help automate this process by grouping the movies by actor; we can then use the `COUNT()` function to count off the number of films in each group. Finally, we can use `ORDER BY` and `LIMIT` to get the top 10 actors by the number of films they've appeared in. Here's the query that does what we want:

```
mysql> SELECT first_name, last_name, COUNT(film_id) AS
    -> actor INNER JOIN film_actor USING (actor_id)
    -> GROUP BY first_name, last_name
    -> ORDER BY num_films DESC LIMIT 5;
```

```
+------------+-------------+-----------+
| first_name | last_name   | num_films |
+------------+-------------+-----------+
| SUSAN      | DAVIS       |        54 |
| GINA       | DEGENERES   |        42 |
| WALTER     | TORN        |        41 |
| MARY       | KEITEL      |        40 |
| MATTHEW    | CARREY      |        39 |
| SANDRA     | KILMER      |        37 |
| SCARLETT   | DAMON       |        36 |
| VAL        | BOLGER      |        35 |
| ANGELA     | WITHERSPOON |        35 |
| UMA        | WOOD        |        35 |
+------------+-------------+-----------+
10 rows in set (0.01 sec)
```

You can see that the output we've asked for is `first_name, last_name, COUNT(film_id) as num_films` , and this tells us exactly what we wanted to know. We group our data by the `first_name` and `last_name` columns, running the `COUNT()` aggregate function in the process. For each "bucket" of rows we got in the previous query, we now get only a single row, albeit giving the information we want. Notice how we've combined `GROUP BY` and `ORDER BY` to get the ordering we wanted: by the number of films, from more to fewer. `GROUP BY` doesn't guarantee ordering, only grouping. Finally, we `LIMIT` the output to 10 rows representing our most productive actors, as otherwise we'd get 199 rows of output.

Let's consider the query further. We'll start with the `GROUP BY` clause. This tells us how to put rows together into groups: in this example, we're telling MySQL that the way to group rows is by `first_name, last_name`. The result is that rows for actors with the same name form a cluster, or bucket—that is, each distinct name becomes one group. Once the rows are grouped, they're treated in the rest of the query as if they're one row. So, for example, when we write `SELECT first_name, last_name`, we get just one row for each group. This is exactly the same as `DISTINCT`, as we've already discussed. The `COUNT()` function tells us about the properties of the group. More specifically, it tells us the number of rows that form each group; you can count any column in a group and you'll get the same answer, so `COUNT(film_id)` is almost always the same as `COUNT(*)` or `COUNT(first_name)`. (See "Aggregate functions" for more details on why we say *almost*.) We could also just do `COUNT(1)`, or in fact specify any literal. Think of this as doing `SELECT 1` from a table and then counting the results. A value of 1 will be output for each row in the table, and `COUNT()` does the counting. One exception is `NULL`: while it's perfectly acceptable and legal to specify `COUNT(NULL)`, the result will always be zero, as `COUNT()` discards `NULL` values. Of course, you can use a column alias for the `COUNT()` column.

Let's try another example. Suppose you want to know how many different actors played in each movie, along with the film name and its category, and get the five films with the largest crews. Here's the query:

```
mysql> SELECT title, name AS category_name, COUNT(*) AS
    -> FROM film INNER JOIN film_actor USING (film_id)
    -> INNER JOIN film_category USING (film_id)
    -> INNER JOIN category USING (category_id)
    -> GROUP BY film_id, category_id
    -> ORDER BY cnt DESC LIMIT 5;
```

```
+------------------+---------------+-----+
| title            | category_name | cnt |
+------------------+---------------+-----+
| LAMBS CINCINATTI | Games         |  15 |
| CRAZY HOME       | Comedy        |  13 |
| CHITTY LOCK      | Drama         |  13 |
| RANDOM GO        | Sci-Fi        |  13 |
| DRACULA CRYSTAL  | Classics      |  13 |
+------------------+---------------+-----+
5 rows in set (0.03 sec)
```

Before we discuss what's new, think about the general function of the query. We join four tables together with `INNER JOIN` using their identifier columns: `film`, `film_actor`, `film_category`, and `category`. Forgetting the aggregation for a moment, the output of this query is one row per combination of movie and actor.

The `GROUP BY` clause puts the rows together into clusters. In this query, we want the films grouped together with their categories. The `GROUP BY` clause uses `film_id` and `category_id` to do that. You can use the `film_id` column from any of the three tables; `film.film_id`, `film_actor.film_id`, and `film_category.film_id` are the same for this purpose. It doesn't matter which one you use; the `INNER JOIN` makes sure they match anyway. The same applies to `category_id`.

As mentioned earlier, even though it's required to list every non-aggregated column in `GROUP BY`, you can `GROUP BY` on columns outside of the `SELECT`. In the previous example query, we're using the `COUNT()` function to tell us how many rows are in each group. For example, you can see that `COUNT(*)` tells us that there are 13 actors in the comedy *CRAZY HOME*. Again, it doesn't matter what column or columns you count in the query: for example, `COUNT(*)` has the same effect as `COUNT(film.film_id)` or `COUNT(category.name)`.

We're then ordering the output by the `COUNT(*)` column aliased `cnt` in descending order and picking the first five rows. Note how there are multiple rows with `cnt` equal to 13. In fact, there are even more of those—six in all—in the database, making this ordering a bit unfair, as movies having the same number of actors will be sorted randomly. You can add another column to the `ORDER BY` clause, like `title`, to make sorting more predictable.

Let's try another example. The `sakila` database isn't only about movies and actors: it's based on movie rentals, after all. We have, among other things, customer information, including data on what films they rented. Say we want to know which customers tend to rent movies from the same category. For example, we might want to adjust our ads based on whether a person likes different film categories or sticks to a single one most of the time. We need to carefully think about our grouping: we don't want to group by movie, as that would just give us the number of times a customer rented it. The resulting query is quite complex, although it's still based around `INNER JOIN` and `GROUP BY`:

```
mysql> SELECT email, name AS category_name, COUNT(categ
    -> FROM customer cs INNER JOIN rental USING (custom
    -> INNER JOIN inventory USING (inventory_id)
    -> INNER JOIN film_category USING (film_id)
    -> INNER JOIN category cat USING (category_id)
    -> GROUP BY 1, 2
    -> ORDER BY 3 DESC LIMIT 5;
```

```
+----------------------------------+---------------+---
| email                            | category_name | cr
+----------------------------------+---------------+---
| WESLEY.BULL@sakilacustomer.org   | Games         |
| ALMA.AUSTIN@sakilacustomer.org   | Animation     |
| KARL.SEAL@sakilacustomer.org     | Animation     |
| LYDIA.BURKE@sakilacustomer.org   | Documentary   |
| NATHAN.RUNYON@sakilacustomer.org | Animation     |
+----------------------------------+---------------+---
5 rows in set (0.08 sec)
```

These customers repeatedly rent films from the same category. What we don't know is if any of them have rented the same movie multiple times, or if those were all different movies within a category. The `GROUP BY` clause hides the details. Again, we use `COUNT(*)` to do the counting of rows in the groups, and you can see the `INNER JOIN` spread over lines 2 to 5 in the query.

The interesting thing about this query is that we didn't explicitly specify column names for the `GROUP BY` or `ORDER BY` clauses. Instead, we used the columns' position numbers (counted from 1) as they appear in the

SELECT clause. This technique saves on typing but can be problematic if you later decide to add another column in the SELECT , which would break the ordering.

As with DISTINCT , there's a danger with GROUP BY that we should mention. Consider the following query:

```
mysql> SELECT COUNT(*) FROM actor GROUP BY first_name,
```

```
+----------+
| COUNT(*) |
+----------+
|        1 |
|        1 |
|      ... |
|        1 |
|        1 |
+----------+
199 rows in set (0.00 sec)
```

It looks simple enough, and it produces the number of times a combination of a given first name and last name was found in the actor table. You might assume that it just outputs 199 rows of the digit 1 . However, if we do a COUNT(*) on the actor table, we get 200 rows. What's the catch? Apparently, two actors have the same first name and last name. These things happen, and you have to be mindful of them. When you group based on columns that do not form a unique identifier, you may accidentally group together unrelated rows, resulting in misleading data. To find the duplicates, we can modify a query that we constructed in "Table Aliases" to look for films with the same name:

```
mysql> SELECT a1.actor_id, a1.first_name, a1.last_name
    -> FROM actor AS a1, actor AS a2
    -> WHERE a1.first_name = a2.first_name
    -> AND a1.last_name = a2.last_name
    -> AND a1.actor_id <> a2.actor_id;
```

```
+----------+------------+-----------+
| actor_id | first_name | last_name |
+----------+-----------+-----------+
|      101 | SUSAN     | DAVIS     |
|      110 | SUSAN     | DAVIS     |
+----------+-----------+-----------+
2 rows in set (0.00 sec)
```

Before we end this section, let's again touch on how MySQL extends the SQL standard around the `GROUP BY` clause. Before MySQL 5.7, it was possible by default to specify an incomplete column list in the `GROUP BY` clause, and, as we've explained, that resulted in a random rows being output within groups for non-grouped dependent columns. For reasons of supporting legacy software, both MySQL 5.7 and My SQL 8.0 continue providing this behavior, though it has to be explicitly enabled. The behavior is controlled by the `ONLY_FULL_GROUP_BY` SQL mode, which is set by default. If you find yourself in a situation where you need to port a program that relies on the legacy `GROUP BY` behavior, we recommend that you do not resort to changing the SQL mode. There are generally two ways to handle this problem. The first is to understand whether the query logic requires incomplete grouping at all—that is rarely the case. The second is to support the random data behavior for non-grouped columns by using either an aggregate function like `MIN()` or `MAX()` or the special `ANY_VALUE()` aggregate function, which, unsurprisingly, just produces a random value from within a group. We'll look more closely at aggregate functions next.

Aggregate functions

We've seen examples of how the `COUNT()` function can be used to tell how many rows are in a group. Here we will cover some other functions commonly used to explore the properties of aggregated rows. We'll also expand a bit on `COUNT()` as it's used frequently:

`COUNT()`

: Returns the number of rows *or* the number of values in a column. Remember we mentioned that `COUNT(*)` is *almost* always the equivalent of `COUNT` *`(<column>)`*. The problem is `NULL`. `COUNT(*)` will do a count of rows returned, regardless of whether the column in those rows is `NULL` or not. However, when you do a `COUNT` *`(<column>)`*, only non-`NULL` values will be counted. For example, in the `sakila` database, a customer's email address may be `NULL`, and we can observe the impact:

```
mysql> SELECT COUNT(*) FROM customer;


+----------+
| count(*) |
+----------+
|      599 |
+----------+
1 row in set (0.00 sec)


mysql> SELECT COUNT(email) FROM customer;


+--------------+
| count(email) |
+--------------+
|          598 |
+--------------+
1 row in set (0.00 sec)
```

We should also add that `COUNT()` can be run with an internal `DISTINCT` clause, as in `COUNT(DISTINCT <column>)`, and will return the number of distinct values instead of all values in this case.

`AVG()`

Returns the average (mean) of the values in the specified column for all rows in a group. For example, you could use it to find the average cost of a house in a city, when the houses are grouped by city:

```
SELECT AVG(cost) FROM house_prices GROUP BY city;
```

`MAX()`

Returns the maximum value from rows in a group. For example, you could use it to find the warmest day in a month, when the rows are grouped by month.

`MIN()`

Returns the minimum value from rows in a group. For example, you could use it to find the youngest student in a class, when the rows are grouped by class.

STD(), *STDDEV()*, *and STDDEV_POP()*

Return the standard deviation of values from rows in a group. For example, you could use these to understand the spread of test scores, when rows are grouped by university course. All three of these are synonyms. `STD()` is a MySQL extension, `STDDEV()` is added for compatibility with Oracle, and `STDDEV_POP()` is a SQL standard function.

SUM()

Returns the sum of values from rows in a group. For example, you could use it to compute the dollar amount of sales in a given month, when rows are grouped by month.

There are other functions available for use with `GROUP BY`, but they're less frequently used than the ones we've introduced here. You can find more details on them in the section on aggregate function descriptions in the MySQL Reference Manual.

The HAVING Clause

You're now familiar with the `GROUP BY` clause, which allows you to sort and cluster data. You should be able to use it to find counts, averages, minimums, and maximums. This section shows how you can use the `HAVING` clause to gain additional control over the aggregation of rows in a `GROUP BY` operation.

Suppose you want to know how many popular actors there are in our database. You've decided to define an actor as popular if they've taken part in at least 40 movies. In the previous section, we tried an almost identical query but without the popularity limitation. We also found that when we grouped the actors by first and last name, we lost one record, so we'll add grouping on the `actor_id` column, which we know to be unique. Here's the new query, with an additional `HAVING` clause that adds the constraint:

```
mysql> SELECT first_name, last_name, COUNT(film_id)
    -> FROM actor INNER JOIN film_actor USING (actor_ic
    -> GROUP BY actor_id, first_name, last_name
    -> HAVING COUNT(film_id) > 40
    -> ORDER BY COUNT(film_id) DESC;
```

```
+------------+-----------+----------------+
| first_name | last_name | COUNT(film_id) |
+------------+-----------+----------------+
| GINA       | DEGENERES |             42 |
| WALTER     | TORN      |             41 |
+------------+-----------+----------------+
2 rows in set (0.01 sec)
```

You can see there are only two actors that meet the new criteria.

The `HAVING` clause must contain an expression or column that's listed in the `SELECT` clause. In this example, we've used `HAVING COUNT(film_id) >= 40`, and you can see that `COUNT(film_id)` is part of the `SELECT` clause. Typically, the expression in the `HAVING` clause uses an aggregate function such as `COUNT()`, `SUM()`, `MIN()`, or `MAX()`. If you find yourself wanting to write a `HAVING` clause that uses a column or expression that isn't in the `SELECT` clause, chances are you should be using a `WHERE` clause instead. The `HAVING` clause is only for deciding how to form each group or cluster, not for choosing rows in the output. We'll show you an example later that illustrates when not to use `HAVING`.

Let's try another example. Suppose you want a list of the top 5 movies that were rented more than 30 times, together with the number of times they were rented, ordered by popularity in reverse. Here's the query you'd use:

```
mysql> SELECT title, COUNT(rental_id) AS num_rented FRO
    -> film INNER JOIN inventory USING (film_id)
    -> INNER JOIN rental USING (inventory_id)
    -> GROUP BY title
    -> HAVING num_rented > 30
    -> ORDER BY num_rented DESC LIMIT 5;
```

```
+--------------------+------------+
| title              | num_rented |
+--------------------+------------+
| BUCKET BROTHERHOOD |         34 |
| ROCKETEER MOTHER    |          33 |
| FORWARD TEMPLE      |          32 |
| GRIT CLOCKWORK      |          32 |
| JUGGLER HARDLY      |          32 |
+---------------------+-------------+
5 rows in set (0.04 sec)
```

You can again see that the expression `COUNT()` is used in both the `SELECT` and `HAVING` clauses. This time, though, we aliased the `COUNT(rental_id)` function to `num_rented` and used the alias in both the `HAVING` and `ORDER BY` clauses.

Now let's consider an example where you shouldn't use `HAVING`. You want to know how many films a particular actor played in. Here's the query you shouldn't use:

```
mysql> SELECT first_name, last_name, COUNT(film_id) AS
    -> actor INNER JOIN film_actor USING (actor_id)
    -> GROUP BY actor_id, first_name, last_name
    -> HAVING first_name = 'EMILY' AND last_name = 'DEE
```

```
+------------+-----------+----------+
| first_name | last_name | film_cnt |
+------------+-----------+----------+
| EMILY      | DEE       |       14 |
+------------+-----------+----------+
1 row in set (0.02 sec)
```

It gets the right answer, but in the wrong—and, for large amounts of data, a much slower—way. It's not the correct way to write the query because the `HAVING` clause isn't being used to decide what rows should form each group but is instead being incorrectly used to filter the answers to display. For this query, we should really use a `WHERE` clause, as follows:

```
mysql> SELECT first_name, last_name, COUNT(film_id) AS
    -> actor INNER JOIN film_actor USING (actor_id)
    -> WHERE first_name = 'EMILY' AND last_name = 'DEE'
    -> GROUP BY actor_id, first_name, last_name;
```

```
+------------+-----------+----------+
| first_name | last_name | film_cnt |
+------------+-----------+----------+
| EMILY      | DEE       |       14 |
+------------+-----------+----------+
1 row in set (0.00 sec)
```

This correct query forms the groups and then picks which groups to display based on the `WHERE` clause.

Advanced Joins

So far in the book, we've used the `INNER JOIN` clause to bring together rows from two or more tables. We'll explain the inner join in more detail in this section, contrasting it with the other join types we discuss: the union, left and right joins, and natural joins. At the conclusion of this section, you'll be able to answer difficult information needs and be familiar with the correct choice of join for the task at hand.

The Inner Join

The `INNER JOIN` clause matches rows between two tables based on the criteria you provide in the `USING` clause. For example, you're very familiar now with an inner join of the `actor` and `film_actor` tables:

```
mysql> SELECT first_name, last_name, film_id FROM
    -> actor INNER JOIN film_actor USING (actor_id)
    -> LIMIT 20;
```

```
+------------+-----------+---------+
| first_name | last_name | film_id |
+------------+-----------+---------+
| PENELOPE   | GUINESS   |       1 |
| PENELOPE   | GUINESS   |      23 |
| ...                              |
| PENELOPE   | GUINESS   |     980 |
| NICK       | WAHLBERG  |       3 |
+------------+-----------+---------+
20 rows in set (0.00 sec)
```

Let's review the key features of an `INNER JOIN`:

- Two tables (or results of a previous join) are listed on either side of the `INNER JOIN` keyphrase.
- The `USING` clause defines one or more columns that are in both tables or results and are used to join or match rows.
- Rows that don't match aren't returned. For example, if you have a row in the `actor` table that doesn't have any matching films in the `film_actor` table, it won't be included in the output.

You can actually write inner-join queries with the `WHERE` clause without using the `INNER JOIN` keyphrase. Here's a rewritten version of the previous query that produces the same result:

```
mysql> SELECT first_name, last_name, film_id
    -> FROM actor, film_actor
    -> WHERE actor.actor_id = film_actor.actor_id
    -> LIMIT 20;
```

```
+------------+-----------+---------+
| first_name | last_name | film_id |
+------------+-----------+---------+
| PENELOPE   | GUINESS   |       1 |
| PENELOPE   | GUINESS   |      23 |
| ...                              |
| PENELOPE   | GUINESS   |     980 |
| NICK       | WAHLBERG  |       3 |
+------------+-----------+---------+
20 rows in set (0.00 sec)
```

You can see that we didn't spell out the inner join: we're selecting from the `actor` and `film_actor` tables the rows where the identifiers match between the tables.

You can modify the `INNER JOIN` syntax to express the join criteria in a way that's similar to using a `WHERE` clause. This is useful if the names of the identifiers don't match between the tables, although that's not the case in this example. Here's the previous query, rewritten in this style:

```
mysql> SELECT first_name, last_name, film_id FROM
    -> actor INNER JOIN film_actor
    -> ON actor.actor_id = film_actor.actor_id
    -> LIMIT 20;
```

```
+------------+-----------+---------+
| first_name | last_name | film_id |
+------------+-----------+---------+
| PENELOPE   | GUINESS   |       1 |
| PENELOPE   | GUINESS   |      23 |
| ...                              |
| PENELOPE   | GUINESS   |     980 |
| NICK       | WAHLBERG  |       3 |
+------------+-----------+---------+
20 rows in set (0.00 sec)
```

You can see that the `ON` clause replaces the `USING` clause and that the columns that follow are fully specified to include the table and column names. If the columns were named differently and uniquely between the two tables, you could omit the table names. There's no real advantage or disadvantage to using `ON` or a `WHERE` clause; it's just a matter of taste. Typically, these days, you'll find most SQL professionals use the `INNER JOIN` with an `ON` clause in preference to `WHERE`, but it's not universal.

Before we move on, let's consider what purpose the `WHERE`, `ON`, and `USING` clauses serve. If you omit the `WHERE` clause from the query we just showed you, you get a very different result. Here's the query and the first few lines of output:

```
mysql> SELECT first_name, last_name, film_id
    -> FROM actor, film_actor LIMIT 20;
```

```
+------------+-------------+---------+
| first_name | last_name   | film_id |
+------------+-------------+---------+
| THORA      | TEMPLE      |       1 |
| JULIA      | FAWCETT     |       1 |
| ...                                |
| DEBBIE     | AKROYD      |       1 |
| MATTHEW    | CARREY      |       1 |
+------------+-------------+---------+
20 rows in set (0.00 sec)
```

The output is nonsensical: what's happened is that each row from the `actor` table has been output alongside each row from the `film_actor` table, for all possible combinations. Since there are 200 actors and 5,462 records in the `film_actor` table, there are 200 × 5,462 = 1,092,400 rows of output, and we know that only 5,462 of those combinations actually make sense (there are only 5,462 records for actors who played in films). We can see the number of rows we'd get without a `LIMIT` with the following query:

```
mysql> SELECT COUNT(*) FROM actor, film_actor;
```

```
+----------+
| COUNT(*) |
+----------+
|  1092400 |
+----------+
1 row in set (0.00 sec)
```

This type of query, without a clause that matches rows, is known as a *Cartesian product*. Incidentally, you also get the Cartesian product if you perform an inner join without specifying a column with a `USING` or `ON` clause, as in this query:

```
SELECT first_name, last_name, film_id
FROM actor INNER JOIN film_actor;
```

In "The Natural Join" we'll introduce the natural join, which is an inner join on identically named columns. While the natural join doesn't use explicitly specified columns, it still produces an inner join, rather than a Cartesian product.

The keyphrase `INNER JOIN` can be replaced with `JOIN` or `STRAIGHT JOIN`; they all do the same thing. However, `STRAIGHT JOIN` forces MySQL to always read the table on the left before it reads the table on the right. We'll have a look at how MySQL processes queries behind the scenes in Chapter 7. The keyphrase `JOIN` is the one you'll see most commonly used: it's a standard shorthand for `INNER JOIN` used by many other database systems besides MySQL, and we will use it in most of our inner-join examples.

The Union

The `UNION` statement isn't really a join operator. Rather, it allows you to combine the output of more than one `SELECT` statement to give a consolidated result set. It's useful in cases where you want to produce a single list from more than one source, or you want to create lists from a single source that are difficult to express in a single query.

Let's look at an example. If you wanted to output all actor *and* movie *and* customer names in the `sakila` database, you could do this with a `UNION` statement. It's a contrived example, but you might want to do this just to list all of the text fragments, rather than to meaningfully present the relationships in the data. There's text in the `actor.first_name` , `film.title` , and `customer.first_name` columns. Here's how to display it:

```
mysql> SELECT first_name FROM actor
    -> UNION
    -> SELECT first_name FROM customer
    -> UNION
    -> SELECT title FROM film;
```

```
+-----------------------------+
| first_name                  |
+-----------------------------+
| PENELOPE                    |
| NICK                        |
| ED                          |
| ...                         |
| ZHIVAGO CORE                |
| ZOOLANDER FICTION           |
| ZORRO ARK                   |
+-----------------------------+
1647 rows in set (0.00 sec)
```

We've shown only a few of the 1,647 rows. The `UNION` statement outputs the results from all the queries together, under a heading appropriate to the first query.

A slightly less contrived example is to create a list of the five most- and least-rented movies in our database. You can do this easily with the `UNION` operator:

```
mysql> (SELECT title, COUNT(rental_id) AS num_rented
    -> FROM film JOIN inventory USING (film_id)
    -> JOIN rental USING (inventory_id)
    -> GROUP BY title ORDER BY num_rented DESC LIMIT 5)
    -> UNION
    -> (SELECT title, COUNT(rental_id) AS num_rented
    -> FROM film JOIN inventory USING (film_id)
    -> JOIN rental USING (inventory_id)
    -> GROUP BY title ORDER BY num_rented ASC LIMIT 5);
```

```
+---------------------+------------+
| title               | num_rented |
+---------------------+------------+
| BUCKET BROTHERHOOD  |         34 |
| ROCKETEER MOTHER    |         33 |
| FORWARD TEMPLE      |         32 |
| GRIT CLOCKWORK      |         32 |
| JUGGLER HARDLY      |         32 |
| TRAIN BUNCH         |          4 |
| HARDLY ROBBERS      |          4 |
| MIXED DOORS         |          4 |
| BUNCH MINDS         |          5 |
| BRAVEHEART HUMAN    |          5 |
+---------------------+------------+
10 rows in set (0.04 sec)
```

The first query uses `ORDER BY` with the `DESC` (descending) modifier and a `LIMIT 5` clause to find the top five most-rented movies. The second query uses `ORDER BY` with the `ASC` (ascending) modifier and a `LIMIT 5` clause to find the five least-rented movies. The `UNION` combines the result sets. Note that there are multiple titles with the same `num_rented` value, and the ordering of titles with the same value is not guaranteed to be determined. You may see different titles listed for `num_rented` values of 32 and 5 on your end.

The `UNION` operator has several limitations:

- The output is labeled with the names of the columns or expressions from the first query. Use column aliases to change this behavior.
- The queries must output the same number of columns. If you try using different numbers of columns, MySQL will report an error.
- All matching columns must have the same type. So, for example, if the first column output from the first query is a date, the first column output from any other query must also be a date.
- The results returned are unique, as if you'd applied a `DISTINCT` to the overall result set. To see this in action, let's try a simple example. Remember we had issues with actors' names—the first name is a bad unique identifier. If we select two actors with the same first name and `UNION` the two queries, we will end up with just one row. The implicit `DISTINCT` operation hides the duplicate (for `UNION`) rows:

```
mysql> SELECT first_name FROM actor WHERE actor_id =
    -> UNION
    -> SELECT first_name FROM actor WHERE actor_id =
```

```
+------------+
| first_name |
+------------+
| KENNETH    |
+------------+
1 row in set (0.01 sec)
```

If you want to show any duplicates, replace `UNION` with `UNION ALL`:

```
mysql> SELECT first_name FROM actor WHERE actor_id =
    -> UNION ALL
    -> SELECT first_name FROM actor WHERE actor_id =
```

```
+------------+
| first_name |
+------------+
| KENNETH    |
| KENNETH    |
+------------+
2 rows in set (0.00 sec)
```

Here, the first name `KENNETH` appears twice.

The implicit `DISTINCT` that `UNION` performs has a nonzero cost on the performance side of things. Whenever you use `UNION`, see whether `UNION ALL` fits logically, and if it can improve query performance.

- If you want to apply `LIMIT` or `ORDER BY` to an individual query that is part of a `UNION` statement, enclose that query in parentheses (as shown in the previous example). It's useful to use parentheses anyway to keep the query easy to understand.

  The `UNION` operation simply concatenates the results of the component queries with no attention to order, so there's not much point in using `ORDER BY` within one of the subqueries. The only time that it makes sense to order a subquery in a `UNION` operation is when you want to select a subset of the results. In our example, we've ordered the movies by the number of times they were rented and then selected only the top five (in the first subquery) and the bottom five (in the second subquery).

  For efficiency, MySQL will actually ignore an `ORDER BY` clause within a subquery if it's used without `LIMIT`. Let's look at some examples to see exactly how this works.

  First, let's run a simple query to list the rental information for a particular movie, along with the time at which the rental happened. We've enclosed the query in parentheses for consistency with our other examples—the parentheses don't actually have any effect here—and haven't used an `ORDER BY` or `LIMIT` clause:

```
mysql> (SELECT title, rental_date, return_date
    -> FROM film JOIN inventory USING (film_id)
    -> JOIN rental USING (inventory_id)
    -> WHERE film_id = 998);
```

```
+--------------+---------------------+-------------
| title        | rental_date         | return_date
+--------------+---------------------+-------------
| ZHIVAGO CORE | 2005-06-17 03:19:20 | 2005-06-21 00
| ZHIVAGO CORE | 2005-07-07 12:18:57 | 2005-07-12 09
| ZHIVAGO CORE | 2005-07-27 14:53:55 | 2005-07-31 19
| ZHIVAGO CORE | 2005-08-20 17:18:48 | 2005-08-26 15
| ZHIVAGO CORE | 2005-05-30 05:15:20 | 2005-06-07 00
| ZHIVAGO CORE | 2005-06-18 06:46:54 | 2005-06-26 09
| ZHIVAGO CORE | 2005-07-12 05:24:02 | 2005-07-16 03
| ZHIVAGO CORE | 2005-08-02 02:05:04 | 2005-08-10 21
| ZHIVAGO CORE | 2006-02-14 15:16:03 | NULL
```

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

The query returns all the times the movie was rented, in no particular order (see the fourth and fifth entries).

Now, let's add an `ORDER BY` clause to this query:

```
mysql> (SELECT title, rental_date, return_date
    -> FROM film JOIN inventory USING (film_id)
    -> JOIN rental USING (inventory_id)
    -> WHERE film_id = 998
    -> ORDER BY rental_date ASC);
```

```
+--------------+---------------------+-------------
| title        | rental_date         | return_date
+--------------+---------------------+-------------
| ZHIVAGO CORE | 2005-05-30 05:15:20 | 2005-06-07 00
| ZHIVAGO CORE | 2005-06-17 03:19:20 | 2005-06-21 00
| ZHIVAGO CORE | 2005-06-18 06:46:54 | 2005-06-26 09
| ZHIVAGO CORE | 2005-07-07 12:18:57 | 2005-07-12 09
| ZHIVAGO CORE | 2005-07-12 05:24:02 | 2005-07-16 03
| ZHIVAGO CORE | 2005-07-27 14:53:55 | 2005-07-31 19
| ZHIVAGO CORE | 2005-08-02 02:05:04 | 2005-08-10 21
| ZHIVAGO CORE | 2005-08-20 17:18:48 | 2005-08-26 15
| ZHIVAGO CORE | 2006-02-14 15:16:03 | NULL
+--------------+---------------------+-------------
9 rows in set (0.00 sec)
```

As expected, we get all the times the movie was rented, in the order of the rental date.

Adding a `LIMIT` clause to the previous query selects the first five rentals, in chronological order—no surprises here:

```
mysql> (SELECT title, rental_date, return_date
    -> FROM film JOIN inventory USING (film_id)
    -> JOIN rental USING (inventory_id)
    -> WHERE film_id = 998
    -> ORDER BY rental_date ASC LIMIT 5);
```

```
+--------------+---------------------+-------------
| title        | rental_date         | return_date
+--------------+---------------------+-------------
| ZHIVAGO CORE | 2005-05-30 05:15:20 | 2005-06-07 00
| ZHIVAGO CORE | 2005-06-17 03:19:20 | 2005-06-21 00
| ZHIVAGO CORE | 2005-06-18 06:46:54 | 2005-06-26 09
| ZHIVAGO CORE | 2005-07-07 12:18:57 | 2005-07-12 09
| ZHIVAGO CORE | 2005-07-12 05:24:02 | 2005-07-16 03
+--------------+---------------------+-------------
5 rows in set (0.01 sec)
```

Now, let's see what happens when we perform a `UNION` operation. In this example, we're using two subqueries, each with an `ORDER BY` clause. We've used a `LIMIT` clause for the second subquery, but not for the first:

```
mysql> (SELECT title, rental_date, return_date
    -> FROM film JOIN inventory USING (film_id)
    -> JOIN rental USING (inventory_id)
    -> WHERE film_id = 998
    -> ORDER BY rental_date ASC)
    -> UNION ALL
    -> (SELECT title, rental_date, return_date
    -> FROM film JOIN inventory USING (film_id)
    -> JOIN rental USING (inventory_id)
    -> WHERE film_id = 998
    -> ORDER BY rental_date ASC LIMIT 5);
```

```
+--------------+---------------------+-------------
| title        | rental_date         | return_date
+--------------+---------------------+-------------
| ZHIVAGO CORE | 2005-06-17 03:19:20 | 2005-06-21 00
| ZHIVAGO CORE | 2005-07-07 12:18:57 | 2005-07-12 09
| ZHIVAGO CORE | 2005-07-27 14:53:55 | 2005-07-31 19
| ZHIVAGO CORE | 2005-08-20 17:18:48 | 2005-08-26 15
| ZHIVAGO CORE | 2005-05-30 05:15:20 | 2005-06-07 00
| ZHIVAGO CORE | 2005-06-18 06:46:54 | 2005-06-26 09
| ZHIVAGO CORE | 2005-07-12 05:24:02 | 2005-07-16 03
| ZHIVAGO CORE | 2005-08-02 02:05:04 | 2005-08-10 21
| ZHIVAGO CORE | 2006-02-14 15:16:03 | NULL
| ZHIVAGO CORE | 2005-05-30 05:15:20 | 2005-06-07 00
| ZHIVAGO CORE | 2005-06-17 03:19:20 | 2005-06-21 00
| ZHIVAGO CORE | 2005-06-18 06:46:54 | 2005-06-26 09
| ZHIVAGO CORE | 2005-07-07 12:18:57 | 2005-07-12 09
| ZHIVAGO CORE | 2005-07-12 05:24:02 | 2005-07-16 03
+--------------+---------------------+--------------
14 rows in set (0.01 sec)
```

As expected, the first subquery returns all the times the movie was rented (the first nine rows of this output), and the second subquery returns the first five rentals (the last five rows of this output). Notice how the first nine rows are not in order (see the fourth and fifth rows), even though the first subquery does have an `ORDER BY` clause. Since we're performing a `UNION` operation, the MySQL server has decided that there's no point sorting the results of the subquery. The second subquery includes a `LIMIT` operation, so the results of that subquery are sorted.

The output of a `UNION` operation isn't guaranteed to be ordered even if the subqueries are ordered, so if you want the final output to be ordered, you should add an `ORDER BY` clause at the end of the whole query. Note that it can be in another order from the subqueries. See the following:

```
mysql> (SELECT title, rental_date, return_date
    -> FROM film JOIN inventory USING (film_id)
    -> JOIN rental USING (inventory_id)
    -> WHERE film_id = 998
    -> ORDER BY rental_date ASC)
    -> UNION ALL
    -> (SELECT title, rental_date, return_date
    -> FROM film JOIN inventory USING (film_id)
    -> JOIN rental USING (inventory_id)
    -> WHERE film_id = 998
    -> ORDER BY rental_date ASC LIMIT 5)
    -> ORDER BY rental_date DESC;
```

```
+--------------+---------------------+--------------
| title        | rental_date         | return_date
+--------------+---------------------+--------------
| ZHIVAGO CORE | 2006-02-14 15:16:03 | NULL
| ZHIVAGO CORE | 2005-08-20 17:18:48 | 2005-08-26 15
| ZHIVAGO CORE | 2005-08-02 02:05:04 | 2005-08-10 21
| ZHIVAGO CORE | 2005-07-27 14:53:55 | 2005-07-31 19
| ZHIVAGO CORE | 2005-07-12 05:24:02 | 2005-07-16 03
| ZHIVAGO CORE | 2005-07-12 05:24:02 | 2005-07-16 03
| ZHIVAGO CORE | 2005-07-07 12:18:57 | 2005-07-12 09
| ZHIVAGO CORE | 2005-07-07 12:18:57 | 2005-07-12 09
| ZHIVAGO CORE | 2005-06-18 06:46:54 | 2005-06-26 09
| ZHIVAGO CORE | 2005-06-18 06:46:54 | 2005-06-26 09
| ZHIVAGO CORE | 2005-06-17 03:19:20 | 2005-06-21 00
| ZHIVAGO CORE | 2005-06-17 03:19:20 | 2005-06-21 00
| ZHIVAGO CORE | 2005-05-30 05:15:20 | 2005-06-07 00
| ZHIVAGO CORE | 2005-05-30 05:15:20 | 2005-06-07 00
+--------------+---------------------+--------------
14 rows in set (0.00 sec)
```

Here’s another example of sorting the final results, including a limit on the number of returned results:

```
mysql> (SELECT first_name, last_name FROM actor WHER
    -> UNION
    -> (SELECT first_name, last_name FROM actor WHER
    -> ORDER BY first_name LIMIT 4;
```

```
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| BELA       | WALKEN    |
| BURT       | TEMPLE    |
| ED         | CHASE     |
| GREGORY    | GOODING   |
+------------+-----------+
4 rows in set (0.00 sec)
```

The `UNION` operation is somewhat unwieldy, and there are generally alternative ways of getting the same result. For example, the previous query could have been written more simply like this:

```
mysql> SELECT first_name, last_name FROM actor
    -> WHERE actor_id < 5 OR actor_id > 190
    -> ORDER BY first_name LIMIT 4;
```

```
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| BELA       | WALKEN    |
| BURT       | TEMPLE    |
| ED         | CHASE     |
| GREGORY    | GOODING    |
+------------+-----------+
4 rows in set (0.00 sec)
```

The Left and Right Joins

The joins we’ve discussed so far output only rows that match between tables. For example, when you join the `film` and `rental` tables through the `inventory` table, you see only the films that were rented. Rows for films that haven’t been rented are ignored. This makes sense in many cases, but it isn’t the only way to join data. This section explains other options you have.

Suppose you do want a comprehensive list of all films and the number of times they’ve been rented. Unlike in the example earlier in this chapter, included in the list you want to see a zero next to movies that haven’t been rented. You can do this with a *left join*, a different type of join that’s driven by one of the two tables participating in the join. In a left join, each row in the left table—the one that’s doing the driving—is processed and output, with the matching data from the second table if it exists and `NULL` values if there is no matching data in the second table. We’ll show you how to write this type of query later in this section, but we’ll start with a simpler example.

Here’s a simple `LEFT JOIN` example. You want to list all movies, and next to each movie you want to show when it was rented. If a movie has never been rented, you want to see that. If it’s been rented many times, you want to see that too. Here’s the query:

```
mysql> SELECT title, rental_date
    -> FROM film LEFT JOIN inventory USING (film_id)
    -> LEFT JOIN rental USING (inventory_id);
```

```
+-----------------------------+---------------------+
| title                       | rental_date         |
+-----------------------------+---------------------+
| ACADEMY DINOSAUR            | 2005-07-08 19:03:15 |
| ACADEMY DINOSAUR            | 2005-08-02 20:13:10 |
| ACADEMY DINOSAUR            | 2005-08-21 21:27:43 |
| ...                                               |
| WAKE JAWS                   | NULL                |
| WALLS ARTIST                | NULL                |
| ...                          |
| ZORRO ARK                    | 2005-07-31 07:32:21 |
| ZORRO ARK                    | 2005-08-19 03:49:28 |
+------------------------------+---------------------+
16087 rows in set (0.06 sec)
```

You can see what happens: movies that have been rented have dates and times, and those that haven't don't (the `rental_date` value is `NULL`). Note also that we `LEFT JOIN` twice in this example. First we join `film` and `inventory`, and we want to make sure that even if a movie is not in our inventory (and thus cannot be rented by definition), we still output it. Then we join the `rental` table with the dataset resulting from the previous join. We use a `LEFT JOIN` again, as we may have films that are not in our inventory, and those won't have any rows in the `rental` table. However, we may also have films listed in our inventory that just haven't been rented. That's why we need to `LEFT JOIN` both tables here.

The order of the tables in the `LEFT JOIN` is important. If you reverse the order in the previous query, you get very different output:

```
mysql> SELECT title, rental_date
    -> FROM rental LEFT JOIN inventory USING (inventory
    -> LEFT JOIN film USING (film_id)
    -> ORDER BY rental_date DESC;
```

```
+------------------------------+---------------------+
| title                        | rental_date         |
+------------------------------+---------------------+
| ...                                                |
| LOVE SUICIDES                | 2005-05-24 23:04:41 |
| GRADUATE LORD                | 2005-05-24 23:03:39 |
| FREAKY POCUS                 | 2005-05-24 22:54:33 |
| BLANKET BEVERLY              | 2005-05-24 22:53:30 |
+------------------------------+---------------------+
16044 rows in set (0.06 sec)
```

In this version, the query is driven by the `rental` table, so all rows from it are matched against the `inventory` table and then against `film`. Since all the rows in the `rental` table by definition are based on the `inventory` table, which is linked to the `film` table, we have no `NULL` values in the output. There can be no rental for a film that doesn't exist. We adjusted the query with `ORDER BY rental_date DESC` to show that we really didn't get any `NULL` values (these would have been last).

By now you can see that left joins are useful when we're sure that our *left* table has some important data, but we're not sure whether the *right* table does. We want to get the rows from the left one with or without the corresponding rows from the right one. Let's try to apply this to a query we wrote in "The GROUP BY Clause", which showed customers renting a lot from the same category. Here's the query, as a reminder:

```
mysql> SELECT email, name AS category_name, COUNT(cat.c
    -> FROM customer cs INNER JOIN rental USING (custom
    -> INNER JOIN inventory USING (inventory_id)
    -> INNER JOIN film_category USING (film_id)
    -> INNER JOIN category cat USING (category_id)
    -> GROUP BY email, category_name
    -> ORDER BY cnt DESC LIMIT 5;
```

```
+----------------------------------+---------------+---
| email                            | category_name | cr
+----------------------------------+---------------+---
| WESLEY.BULL@sakilacustomer.org   | Games         |
| ALMA.AUSTIN@sakilacustomer.org   | Animation     |
| KARL.SEAL@sakilacustomer.org     | Animation     |
| LYDIA.BURKE@sakilacustomer.org   | Documentary   |
| NATHAN.RUNYON@sakilacustomer.org | Animation     |
+----------------------------------+---------------+---
5 rows in set (0.06 sec)
```

What if we now want to see whether a customer we found this way rents films from anything but their favorite category? It turns out that's actually pretty difficult!

Let's consider this task. We need to start with the `category` table, as that will have all the categories we have for our films. We then need to start constructing a whole chain of left joins. First we left join `category` to `film_category`, as we may have categories with no films. Then we left join the result to the `inventory` table, as some movies we know about may not be in our catalog. We then left join that result to the `rental` table, as customers may not have rented some of the films in a category. Finally, we need to left join that result to our `customer` table. Even though there can be no associated customer record without a rental, omitting the left join here will cause MySQL to discard rows for categories that end up with no customer records.

Now, after this whole long explanation, can we finally go ahead and filter by email address and get our data? No! Unfortunately, by adding a `WHERE` condition on the table that is not *left* in our left-join relationship, we break the idea of this join. See what happens:

```
mysql> SELECT COUNT(*) FROM category;


+----------+
| COUNT(*) |
+----------+
|       16 |
+----------+
1 row in set (0.00 sec)


mysql> SELECT email, name AS category_name, COUNT(categ
    -> FROM category cat LEFT JOIN film_category USING
    -> LEFT JOIN inventory USING (film_id)
    -> LEFT JOIN rental USING (inventory_id)
    -> JOIN customer cs ON rental.customer_id = cs.cust
    -> WHERE cs.email = 'WESLEY.BULL@sakilacustomer.org
    -> GROUP BY email, category_name
    -> ORDER BY cnt DESC;
```

```
+--------------------------------+---------------+-----
| email                          | category_name | cnt
+--------------------------------+---------------+-----
| WESLEY.BULL@sakilacustomer.org | Games         |   9
| WESLEY.BULL@sakilacustomer.org | Foreign       |   6
| ...
| WESLEY.BULL@sakilacustomer.org | Comedy        |   1
| WESLEY.BULL@sakilacustomer.org | Sports        |   1
+--------------------------------+---------------+-----
14 rows in set (0.00 sec)
```

We got 14 categories for our customer, while there are 16 in total. In fact, MySQL will optimize away all the left joins in this query, as it understands they are meaningless when put like this. There's no easy way to answer the question we have with just joins—we'll get back to this example in "Nested Queries in JOINs".

The query that we've written is still useful, though. While by default `sakila` does not have a film category in which no films have been rented, if we expand our database slightly, we can see the effectiveness of left joins:

```
mysql> INSERT INTO category(name) VALUES ('Thriller');
```

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

```
mysql> SELECT cat.name, COUNT(rental_id) cnt
    -> FROM category cat LEFT JOIN film_category USING
    -> LEFT JOIN inventory USING (film_id)
    -> LEFT JOIN rental USING (inventory_id)
    -> LEFT JOIN customer cs ON rental.customer_id = cs
    -> GROUP BY 1
    -> ORDER BY 2 DESC;
```

```
+---------------+------+
| category_name | cnt  |
+---------------+------+
| Sports        | 1179 |
| Animation     | 1166 |
| ...                  |
| Music         |  830 |
| Thriller      |    0 |
+---------------+------+
17 rows in set (0.07 sec)
```

If we were to use a regular `INNER JOIN` (or just `JOIN`, its synonym) here, we wouldn't get information for the Thriller category, and we might get different counts for other categories. As `category` is our leftmost table, it drives the process of the query, and every row from that table is present in the output.

We’ve shown you that it matters what comes before and after the `LEFT JOIN` keyphrase. Whatever is on the left drives the process, hence the name “left join.” If you really don’t want to reorganize your query so it matches that template, you can use `RIGHT JOIN`. It’s exactly the same, except whatever is on the right drives the process.

Earlier we showed the importance of the order of the tables in a left join using two queries for film rental information. Let’s rewrite the second of them (which showed incorrect data) using a right join:

```
mysql> SELECT title, rental_date
    -> FROM rental RIGHT JOIN inventory USING (inventor
    -> RIGHT JOIN film USING (film_id)
    -> ORDER BY rental_date DESC;
```

```
...
| SUICIDES SILENCE             | NULL                |
| TADPOLE PARK                 | NULL                |
| TREASURE COMMAND             | NULL                |
| VILLAIN DESPERATE            | NULL                |
| VOLUME HOUSE                 | NULL                |
| WAKE JAWS                    | NULL                |
| WALLS ARTIST                 | NULL                |
+------------------------------+---------------------+
16087 rows in set (0.06 sec)
```

We got the same number of rows, and we can see that the `NULL` values are the same as those the “correct” query gave us. The right join is useful sometimes because it allows you to write a query more naturally, expressing it in a way that’s more intuitive. However, you won’t often see it used, and we’d recommend avoiding it where possible.

Both left and right joins can use the `USING` and `ON` clauses discussed in “The Inner Join”. You should use one or the other: without them you’ll get the Cartesian product, as discussed in that section.

There’s also an extra `OUTER` keyword that you can optionally use in left and right joins to make them read as `LEFT OUTER JOIN` and `RIGHT OUTER`

JOIN . It’s just an alternative syntax that doesn’t do anything different, and you won’t often see it used. We stick to the basic versions in this book.

The Natural Join

We’re not big fans of the natural join that we describe in this section. It’s included here only for completeness, and because you’ll see it used sometimes in SQL statements you’ll encounter. Our advice is to avoid using it where possible.

A natural join is, well, supposed to be magically natural. This means that you tell MySQL what tables you want to join, and it figures out how to do it and gives you an `INNER JOIN` result set. Here’s an example for the `actor_info` and `film_actor` tables:

```
mysql> SELECT first_name, last_name, film_id
    -> FROM actor_info NATURAL JOIN film_actor
    -> LIMIT 20;
```

```
+------------+-----------+---------+
| first_name | last_name | film_id |
+------------+-----------+---------+
| PENELOPE   | GUINESS   |       1 |
| PENELOPE   | GUINESS   |      23 |
| ...                              |
| PENELOPE   | GUINESS   |     980 |
| NICK       | WAHLBERG  |       3 |
+------------+-----------+---------+
20 rows in set (0.28 sec)
```

In reality, it’s not quite magical: all MySQL does is look for columns with the same names and, behind the scenes, add these silently into an inner join with join conditions written into the `WHERE` clause. So, the previous query is actually translated into something like this:

```
mysql> SELECT first_name, last_name, film_id FROM
    -> actor_info JOIN film_actor
    -> WHERE (actor_info.actor_id = film_actor.actor_ic
    -> LIMIT 20;
```

If the identifier columns don't share the same name, natural joins won't work. Also, more dangerously, if columns that do share the same names aren't identifiers, they'll get thrown into the behind-the-scenes `USING` clause anyway. You can very easily see this in the `sakila` database. In fact, that's why we resorted to showing the preceding example with `actor_info`, which isn't even a table: it's a view. Let's see what would have happened if we used the regular `actor` and `film_actor` tables:

```
mysql> SELECT first_name, last_name, film_id FROM actor
```

```
Empty set (0.01 sec)
```

But how? The problem is: `NATURAL JOIN` really does take *all* of the columns into consideration. With the `sakila` database, that's a huge roadblock, as every table has a `last_update` column. If you were to run an `EXPLAIN` statement on the previous query and then execute `SHOW WARNINGS`, you'd see that the resulting query is meaningless:

```
mysql> SHOW WARNINGS\G
```

```
*************************** 1. row ********************
  Level: Note
   Code: 1003
Message: /* select#1 */ select `sakila`.`customer`.`ema
`sakila`.`rental`.`rental_date` AS `rental_date`
from `sakila`.`customer` join `sakila`.`rental`
where ((`sakila`.`rental`.`last_update` = `sakila`.`cus
and (`sakila`.`rental`.`customer_id` = `sakila`.`custon
1 row in set (0.00 sec)
```

You'll sometimes see the natural join mixed with left and right joins. The following are valid join syntaxes: `NATURAL LEFT JOIN`, `NATURAL LEFT OUTER JOIN`, `NATURAL RIGHT JOIN`, and `NATURAL RIGHT OUTER JOIN`. The former two are left joins without `ON` or `USING` clauses, and the latter two are right joins. Again, avoid writing them when you can, but you should understand what they mean if you see them used.

Constant Expressions in Joins

In all of the examples of joins we've given you so far, we've used column identifiers to define the join condition. When you're using the `USING` clause, that's the only possible way to go. When you're defining the join conditions in a `WHERE` clause, that's also the only thing that will work. However, when you're using the `ON` clause, you can actually add constant expressions.

Let's consider an example, listing all films for a particular actor:

```
mysql> SELECT first_name, last_name, title
    -> FROM actor JOIN film_actor USING (actor_id)
    -> JOIN film USING (film_id)
    -> WHERE actor_id = 11;
```

```
+------------+-----------+--------------------+
| first_name | last_name | title              |
+------------+-----------+--------------------+
| ZERO       | CAGE      | CANYON STOCK       |
| ZERO       | CAGE      | DANCES NONE        |
| ...                                         |
| ZERO       | CAGE      | WEST LION          |
| ZERO       | CAGE      | WORKER TARZAN      |
+------------+-----------+--------------------+
25 rows in set (0.00 sec)
```

We can move the `actor_id` clause into the join like this:

```
mysql> SELECT first_name, last_name, title
    -> FROM actor JOIN film_actor
    ->   ON actor.actor_id = film_actor.actor_id
    ->   AND actor.actor_id = 11
    -> JOIN film USING (film_id);
```

```
+------------+-----------+--------------------+
| first_name | last_name | title              |
+------------+-----------+--------------------+
| ZERO       | CAGE      | CANYON STOCK       |
| ZERO       | CAGE      | DANCES NONE        |
| ...                                         |
| ZERO       | CAGE      | WEST LION          |
| ZERO        | CAGE       | WORKER TARZAN      |
+-----------+----------+--------------------+
25 rows in set (0.00 sec)
```

Well, that's neat, of course, but why? Is this any more expressive than having the proper `WHERE` clause? The answer to both questions is that constant conditions in joins are evaluated and resolved differently than the conditions in the `WHERE` clause are. It's easier to show this with an example, but the preceding query is a poor one. The impact of constant conditions in joins is best shown with a left join.

Remember this query from the section on left joins:

```
mysql> SELECT email, name AS category_name, COUNT(renta
    -> FROM category cat LEFT JOIN film_category USING
    -> LEFT JOIN inventory USING (film_id)
    -> LEFT JOIN rental USING (inventory_id)
    -> LEFT JOIN customer cs USING (customer_id)
    -> WHERE cs.email = 'WESLEY.BULL@sakilacustomer.org
    -> GROUP BY email, category_name
    -> ORDER BY cnt DESC;
```

```
+--------------------------------+--------------+-----
| email                          | category_name | cnt
+--------------------------------+--------------+-----
| WESLEY.BULL@sakilacustomer.org | Games         |   9
| WESLEY.BULL@sakilacustomer.org | Foreign       |   6
| ...
| WESLEY.BULL@sakilacustomer.org | Comedy        |   1
| WESLEY.BULL@sakilacustomer.org | Sports        |   1
+--------------------------------+--------------+-----
14 rows in set (0.01 sec)
```

If we go ahead and move the `cs.email` clause to the `LEFT JOIN customer cs` part, we'll see completely different results:

```
mysql> SELECT email, name AS category_name, COUNT(renta
    -> FROM category cat LEFT JOIN film_category USING
    -> LEFT JOIN inventory USING (film_id)
    -> LEFT JOIN rental USING (inventory_id)
    -> LEFT JOIN customer cs ON rental.customer_id = cs
    -> AND cs.email = 'WESLEY.BULL@sakilacustomer.org'
    -> GROUP BY email, category_name
    -> ORDER BY cnt DESC;
```

```
+--------------------------------+-------------+-------+
| email                          | name        | cnt   |
+--------------------------------+-------------+-------+
| NULL                           | Sports      | 1178  |
| NULL                           | Animation   | 1164  |
| ...                                                  |
| NULL                           | Travel      |   834 |
| NULL                           | Music       |   829 |
| WESLEY.BULL@sakilacustomer.org | Games       |     9 |
| WESLEY.BULL@sakilacustomer.org | Foreign     |     6 |
| ...                                                  |
| WESLEY.BULL@sakilacustomer.org | Comedy      |     1 |
| NULL                           | Thriller    |     0 |
+--------------------------------+-------------+-------+
31 rows in set (0.07 sec)
```

That's interesting! Instead of getting only Wesley's rental counts per category, we also get rental counts for everyone else broken down by category. That even includes our new and so far empty Thriller category. Let's try to understand what happens here.

The `WHERE` clause's contents are applied logically after the joins are resolved and executed. We tell MySQL we only need rows from whatever we join where the `cs.email` column equals `'WESLEY.BULL@sakilacustomer.org'`. In reality, MySQL is smart enough to optimize this situation and will actually start the plan execution as if regular inner joins were used. When we have the `cs.email` condition within the `LEFT JOIN customer` clause, we tell MySQL that we want to add columns from the `customer` table to our result set so far (which includes the `category`, `inventory`, and `rental` tables), but only when the certain value is present in the `email` column. Since this is a `LEFT JOIN`, we get `NULL` in every column of `customer` in rows that didn't match.

It's important to be aware of this behavior.

Nested Queries

Nested queries, supported by MySQL since version 4.1, are the most difficult to learn. However, they provide a powerful, useful, and concise way of expressing difficult information needs in short SQL statements. This section explains them, beginning with simple examples and leading to the more complex features of the `EXISTS` and `IN` statements. At the conclusion of this section, you'll have completed everything this book contains about querying data, and you should understand almost any SQL query you encounter.

Nested Query Basics

You know how to find the names of all the actors who played in a particular movie using an `INNER JOIN`:

```
mysql> SELECT first_name, last_name FROM
    -> actor JOIN film_actor USING (actor_id)
    -> JOIN film USING (film_id)
    -> WHERE title = 'ZHIVAGO CORE';
```

```
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| UMA        | WOOD      |
| NICK       | STALLONE  |
| GARY       | PENN      |
| SALMA      | NOLTE     |
| KENNETH    | HOFFMAN   |
| WILLIAM    | HACKMAN   |
+------------+-----------+
6 rows in set (0.00 sec)
```

But there's another way, using a *nested query*:

```
mysql> SELECT first_name, last_name FROM
    -> actor JOIN film_actor USING (actor_id)
    -> WHERE film_id = (SELECT film_id FROM film
    -> WHERE title = 'ZHIVAGO CORE');
```

```
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| UMA        | WOOD      |
| NICK       | STALLONE  |
| GARY       | PENN      |
| SALMA      | NOLTE     |
| KENNETH    | HOFFMAN   |
| WILLIAM    | HACKMAN   |
+------------+-----------+
6 rows in set (0.00 sec)
```

It's called a nested query because one query is inside another. The *inner query*, or *subquery*—the one that is nested—is written in parentheses, and you can see that it determines the `film_id` for the film with the title `ZHIVAGO CORE`. The parentheses are required for inner queries. The *outer query* is the one that's listed first and isn't parenthesized here: you can see that it finds the `first_name` and `last_name` of the actors from a `JOIN` with `film_actor` with a `film_id` that matches the result of the subquery. So, overall, the inner query finds the `film_id`, and the outer query uses it to find actors' names. Whenever nested queries are used, it's possible to rewrite them as a few separate queries. Let's do that with the previous example, as it may help you understand what is going on:

```
mysql> SELECT film_id FROM film WHERE title = 'ZHIVAGO
```

```
+---------+
| film_id |
+---------+
|     998 |
+---------+
1 row in set (0.03 sec)
```

```
mysql> SELECT first_name, last_name
    -> FROM actor JOIN film_actor USING (actor_id)
    -> WHERE film_id = 998;
```

```
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| UMA        | WOOD      |
| NICK       | STALLONE  |
| GARY       | PENN      |
| SALMA      | NOLTE     |
| KENNETH    | HOFFMAN   |
| WILLIAM    | HACKMAN   |
+------------+-----------+
6 rows in set (0.00 sec)
```

So, which approach is preferable: nested or not nested? The answer isn't easy. In terms of performance, the answer is usually *not*: nested queries are hard to optimize, so they're almost always slower to run than the unnested alternative.

Does this mean you should avoid nesting? The answer is no: sometimes it's your only choice if you want to write a single query, and sometimes nested queries can answer information needs that can't be easily solved otherwise. What's more, nested queries are expressive. Once you're comfortable with the idea, they're a very readable way to show how a query is evaluated. In fact, many SQL designers advocate teaching nested queries before the join-based alternatives we've shown you in the past few sections. We'll show you examples where nesting is readable and powerful throughout this section.

Before we begin to cover the keywords that can be used in nested queries, let's take a look at an example that can't be done easily in a single query—at least, not without MySQL's nonstandard, although ubiquitous, `LIMIT` clause! Suppose you want to know what movie a customer rented most recently. To do this, following the methods we've learned previously, you could find the date and time of the most recently stored row in the `rental` table for that customer:

```
mysql> SELECT MAX(rental_date) FROM rental
    -> JOIN customer USING (customer_id)
    -> WHERE email = 'WESLEY.BULL@sakilacustomer.org';
```

```
+---------------------+
| MAX(rental_date)    |
+---------------------+
| 2005-08-23 15:46:33 |
+---------------------+
1 row in set (0.01 sec)
```

You can then use the output as input to another query to find the film title:

```
mysql> SELECT title FROM film
    -> JOIN inventory USING (film_id)
    -> JOIN rental USING (inventory_id)
    -> JOIN customer USING (customer_id)
    -> WHERE email = 'WESLEY.BULL@sakilacustomer.org'
    -> AND rental_date = '2005-08-23 15:46:33';
```

```
+-------------+
| title       |
+-------------+
| KARATE MOON |
+-------------+
1 row in set (0.00 sec)
```

NOTE

In "User Variables" we'll show you how you can use variables to avoid having to type in the value in the second query.

With a nested query, you can do both steps in one shot:

```
mysql> SELECT title FROM film JOIN inventory USING (fil
    -> JOIN rental USING (inventory_id)
    -> WHERE rental_date = (SELECT MAX(rental_date) FRO
    -> JOIN customer USING (customer_id)
    -> WHERE email = 'WESLEY.BULL@sakilacustomer.org');
```

```
+-------------+
| title       |
+-------------+
| KARATE MOON |
+-------------+
1 row in set (0.01 sec)
```

You can see the nested query combines the two previous queries. Rather than using the constant date and time value discovered from a previous query, it executes the query directly as a subquery. This is the simplest type of nested query, one that returns a *scalar operand*--that is, a single value.

TIP

The previous example used the equality operator (`=`). You can use all types of comparison operators: `<` (less than), `<=` (less than or equal to), `>` (greater than), `>=` (greater than or equal to), and `!=` (not equals) or `<>` (not equals).

The ANY, SOME, ALL, IN, and NOT IN Clauses

Before we start to show some more advanced features of nested queries, we need to switch to a new database in our examples. Unfortunately, the `sakila` database is a little too well normalized to effectively demonstrate the full power of nested querying. So, let's add a new database to give us something to play with.

The database we'll install is the `employees` sample database. You can find instructions for installation in the MySQL documentation or in the database's GitHub repo. Either clone the repository using `git` or download the latest release (1.0.7 at the time of writing). Once you have the necessary files ready, you need to run two commands.

The first command creates the necessary structures and loads the data:

```
$ mysql -uroot -p < employees.sql
INFO
CREATING DATABASE STRUCTURE
INFO
storage engine: InnoDB
INFO
LOADING departments
INFO
LOADING employees
INFO
LOADING dept_emp
INFO
LOADING dept_manager
INFO
```

```
LOADING titles
INFO
LOADING salaries
data_load_time_diff
00:00:28
```

The second command verifies the installation:

```
$ mysql -uroot -p < test_employees_md5.sql
INFO
TESTING INSTALLATION
table_name      expected_records        expected_crc
departments     9       d1af5e170d2d1591d776d5638d71fc5
dept_emp        331603  ccf6fe516f990bdaa49713fc478701b
dept_manager    24      8720e2f0853ac9096b689c14664f847
employees       300024  4ec56ab5ba37218d187cf6ab09ce1aa
salaries        2844047 fd220654e95aea1b169624ffe3fca93
titles  443308  bfa016c472df68e70a03facafa1bc0a8
table_name      found_records           found_crc
departments     9       d1af5e170d2d1591d776d5638d71fc5
dept_emp        331603  ccf6fe516f990bdaa49713fc478701b
dept_manager    24      8720e2f0853ac9096b689c14664f847
employees       300024  4ec56ab5ba37218d187cf6ab09ce1aa
salaries        2844047 fd220654e95aea1b169624ffe3fca93
titles  443308  bfa016c472df68e70a03facafa1bc0a8
table_name      records_match   crc_match
departments     OK      ok
dept_emp        OK      ok
dept_manager    OK      ok
employees       OK      ok
salaries        OK      ok
titles  OK      ok
computation_time
00:00:25
summary result
CRC     OK
count   OK
```

Once this is done, you can proceed to work through the examples we'll be providing next.

To connect to the new database, either run `mysql` from the command line like this (or specify `employees` as a target for your MySQL client of choice):

```
$ mysql employees
```

Or execute the following at a `mysql` prompt to change the default database:

```
mysql> USE employees
```

Now you’re ready to move forward.

Using ANY and IN

Now that you’ve created the sample tables, you can try an example using `ANY`. Suppose you’re looking to find assistant engineers who’ve been working longer than the least experienced manager. You can express this information need as follows:

```
mysql> SELECT emp_no, first_name, last_name, hire_date
    -> FROM employees JOIN titles USING (emp_no)
    -> WHERE title = 'Assistant Engineer'
    -> AND hire_date < ANY (SELECT hire_date FROM
    -> employees JOIN titles USING (emp_no)
    -> WHERE title = 'Manager');
```

```
+--------+----------------+------------------+---------
| emp_no | first_name     | last_name        | hire_dat
+--------+----------------+------------------+---------
|  10009 | Sumant         | Peac             | 1985-02-
|  10066 | Kwee           | Schusler         | 1986-02-
| ...
| ...
| 499958 | Srinidhi       | Theuretzbacher   | 1989-12-
| 499974 | Shuichi        | Piazza           | 1989-09-
+--------+----------------+------------------+---------
10747 rows in set (0.20 sec)
```

Turns out there are a lot of people who meet these criteria! The subquery finds the dates on which managers were hired:

```
mysql> SELECT hire_date FROM
    -> employees JOIN titles USING (emp_no)
    -> WHERE title = 'Manager';
```

```
+------------+
| hire_date  |
+------------+
| 1985-01-01 |
| 1986-04-12 |
| ...        |
| 1991-08-17 |
| 1989-07-10 |
+------------+
24 rows in set (0.10 sec)
```

The outer query goes through each employee with the title `Associate Engineer`, returning the engineer if their hire date is lower (older) than any of the values in the set returned by the subquery. So, for example, `Sumant Peac` is output because `1985-02-18` is older than at least one value in the set (as you can see, the second hire date returned for managers is `1986-04-12`). The `ANY` keyword means just that: it's true if the column or expression preceding it is true for *any* of the values in the set returned by the subquery. Used in this way, `ANY` has the alias `SOME`, which was included so that some queries can be read more clearly as English expressions; it doesn't do anything different, and you'll rarely see it used.

The `ANY` keyword gives you more power in expressing nested queries. Indeed, the previous query is the first nested query in this section with a *column subquery*—that is, the results returned by the subquery are one or more values from a column, instead of a single scalar value as in the previous section. With this, you can now compare a column value from an outer query to a set of values returned from a subquery.

Consider another example using `ANY`. Suppose you want to know the managers who also have some other title. You can do this with the following nested query:

```
mysql> SELECT emp_no, first_name, last_name
    -> FROM employees JOIN titles USING (emp_no)
    -> WHERE title = 'Manager'
    -> AND emp_no = ANY (SELECT emp_no FROM employees
    -> JOIN titles USING (emp_no) WHERE
    -> title <> 'Manager');
```

```
+--------+------------+------------+
| emp_no | first_name | last_name  |
+--------+------------+------------+
| 110022 | Margareta  | Markovitch |
| 110039 | Vishwani   | Minakawa   |
| ...                              |
| 111877 | Xiaobin    | Spinelli   |
| 111939 | Yuchang    | Weedman    |
+--------+------------+------------+
24 rows in set (0.11 sec)
```

The `= ANY` causes the outer query to return a manager when the `emp_no` is equal to any of the engineer employee numbers returned by the subquery. The `= ANY` keyphrase has the alias `IN`, which you'll see commonly used in nested queries. Using `IN`, the previous example can be rewritten as:

```
mysql> SELECT emp_no, first_name, last_name
    -> FROM employees JOIN titles USING (emp_no)
    -> WHERE title = 'Manager'
    -> AND emp_no IN (SELECT emp_no FROM employees
    -> JOIN titles USING (emp_no) WHERE
    -> title <> 'Manager');
```

```
+--------+------------+------------+
| emp_no | first_name | last_name  |
+--------+------------+------------+
| 110022 | Margareta  | Markovitch |
| 110039 | Vishwani   | Minakawa   |
| ...                              |
| 111877 | Xiaobin    | Spinelli   |
| 111939 | Yuchang    | Weedman    |
+--------+------------+------------+
24 rows in set (0.11 sec)
```

Of course, for this particular example, you could also have used a join query. Note that we have to use `DISTINCT` here, because otherwise we get 30 rows returned. Some people hold more than one non-engineer title:

```
mysql> SELECT DISTINCT emp_no, first_name, last_name
    -> FROM employees JOIN titles mgr USING (emp_no)
    -> JOIN titles nonmgr USING (emp_no)
    -> WHERE mgr.title = 'Manager'
    -> AND nonmgr.title <> 'Manager';
```

```
+--------+------------+-------------+
| emp_no | first_name | last_name   |
+--------+------------+-------------+
| 110022 | Margareta  | Markovitch  |
| 110039 | Vishwani   | Minakawa    |
| ...                               |
| 111877 | Xiaobin    | Spinelli    |
| 111939 | Yuchang    | Weedman     |
+--------+------------+-------------+
24 rows in set (0.11 sec)
```

Again, nested queries are expressive but typically slow in MySQL, so use a join where you can.

Using ALL

Suppose you want to find assistant engineers who are more experienced than all of the managers—that is, more experienced than the most experienced manager. You can do this with the `ALL` keyword in place of `ANY` :

```
mysql> SELECT emp_no, first_name, last_name, hire_date
    -> FROM employees JOIN titles USING (emp_no)
    -> WHERE title = 'Assistant Engineer'
    -> AND hire_date < ALL (SELECT hire_date FROM
    -> employees JOIN titles USING (emp_no)
    -> WHERE title = 'Manager');
```

```
Empty set (0.18 sec)
```

You can see that there are no answers. We can inspect the data further to check what is the oldest hire date of a manager and of an assistant engineer:

```
mysql> (SELECT 'Assistant Engineer' AS title,
    -> MIN(hire_date) AS mhd FROM employees
    -> JOIN titles USING (emp_no)
    -> WHERE title = 'Assistant Engineer')
    -> UNION
    -> (SELECT 'Manager' title, MIN(hire_date) mhd FROM
    -> JOIN titles USING (emp_no)
    -> WHERE title = 'Manager');
```

```
+--------------------+------------+
| title              | mhd        |
+--------------------+------------+
| Assistant Engineer | 1985-02-01 |
| Manager            | 1985-01-01 |
+--------------------+------------+
2 rows in set (0.26 sec)
```

Looking at the data, we see that the first manager was hired on January 1, 1985, and the first assistant engineer only on February 1 of the same year. While the `ANY` keyword returns values that satisfy at least one condition (Boolean OR), the `ALL` keyword returns values only where all the conditions are satisfied (Boolean AND).

We can use the alias `NOT IN` in place of `<> ANY` or `!= ANY`. Let's find all the managers who aren't senior staff:

```
mysql> SELECT emp_no, first_name, last_name
    -> FROM employees JOIN titles USING (emp_no)
    -> WHERE title = 'Manager' AND emp_no NOT IN
    -> (SELECT emp_no FROM titles
    -> WHERE title = 'Senior Staff');
```

```
+--------+------------+-------------+
| emp_no | first_name | last_name   |
+--------+------------+-------------+
| 110183 | Shirish    | Ossenbruggen |
| 110303 | Krassimir  | Wegerle     |
| ...                               |
| 111400 | Arie       | Staelin     |
| 111692 | Tonny      | Butterworth |
+--------+------------+-------------+
15 rows in set (0.09 sec)
```

As an exercise, try writing this query using the `ANY` syntax and as a join query.

The `ALL` keyword has a few tricks and traps:

- If it's false for any value, it's false. Suppose that table `a` contains a row with the value 14, and table `b` contains the values 16, 1, and `NULL`. If you check whether the value in `a` is greater than `ALL` values in `b`, you'll get `false`, since 14 isn't greater than 16. It doesn't matter that the other values are 1 and `NULL`.
- If it isn't false for any value, it isn't true unless it's true for all values. Suppose that table `a` again contains 14, and `b` contains 1 and `NULL`. If you check whether the value in `a` is greater than `ALL` values in `b`, you'll get `UNKNOWN` (neither true nor false) because it can't be determined whether `NULL` is greater than or less than 14.
- If the table in the subquery is empty, the result is always true. Hence, if `a` contains 14 and `b` is empty, you'll get `true` when you check if the value in `a` is greater than `ALL` values in `b`.

When using the `ALL` keyword, be very careful with tables that can have `NULL` values in columns; consider disallowing `NULL` values in such cases. Also, be careful with empty tables.

Writing row subqueries

In the previous examples, the subquery returned a single scalar value (such as an `actor_id`) or a set of values from one column (such as all of the `emp_no` values). This section describes another type of subquery, the *row subquery*, that works with multiple columns from multiple rows.

Suppose you're interested in whether a manager had another position within the same calendar year. To answer this need, you must match both the employee number and the title assignment date, or, more precisely, year. You can write this as a join:

```
mysql> SELECT mgr.emp_no, YEAR(mgr.from_date) AS fd
    -> FROM titles AS mgr, titles AS other
    -> WHERE mgr.emp_no = other.emp_no
    -> AND mgr.title = 'Manager'
    -> AND mgr.title <> other.title
    -> AND YEAR(mgr.from_date) = YEAR(other.from_date);
```

```
+--------+------+
| emp_no | fd   |
+--------+------+
| 110765 | 1989 |
| 111784 | 1988 |
+--------+------+
2 rows in set (0.11 sec)
```

But you can also write it as a nested query:

```
mysql> SELECT emp_no, YEAR(from_date) AS fd
    -> FROM titles WHERE title = 'Manager' AND
    -> (emp_no, YEAR(from_date)) IN
    -> (SELECT emp_no, YEAR(from_date)
    -> FROM titles WHERE title <> 'Manager');
```

```
+--------+------+
| emp_no | fd   |
+--------+------+
| 110765 | 1989 |
| 111784 | 1988 |
+--------+------+
2 rows in set (0.12 sec)
```

You can see there's a different syntax being used in this nested query: a list of two column names in parentheses follows the `WHERE` statement, and the inner query returns two columns. We'll explain this syntax next.

The row subquery syntax allows you to compare multiple values per row. The expression `(emp_no, YEAR(from_date))` means two values per row are compared to the output of the subquery. You can see following the `IN` keyword that the subquery returns two values, `emp_no` and `YEAR(from_date)`. So, the fragment:

```
(emp_no, YEAR(from_date)) IN (SELECT emp_no, YEAR(from_
FROM titles WHERE title <> 'Manager')
```

matches manager numbers and starting years to nonmanager numbers and starting years, and returns a true value when a match is found. The result is that if a matching pair is found, the overall query outputs a result. This is a typical row subquery: it finds rows that exist in two tables.

To explain the syntax further, let's consider another example. Suppose you want to see if a particular employee is a senior staff member. You can do this with the following query:

```
mysql> SELECT first_name, last_name
    -> FROM employees, titles
    -> WHERE (employees.emp_no, first_name, last_name,
    -> (titles.emp_no, 'Marjo', 'Giarratana', 'Senior S
```

```
+------------+------------+
| first_name | last_name  |
+------------+------------+
| Marjo      | Giarratana |
+------------+------------+
1 row in set (0.09 sec)
```

It's not a nested query, but it shows you how the new row subquery syntax works. You can see that the query matches the list of columns before the equals sign, `(employees.emp_no, first_name, last_name, title)`, to the list of columns and values after the equals sign, `(titles.emp_no, 'Marjo', 'Giarratana', 'Senior Staff')`. So, when the `emp_no` values match, the employee's full name is `Marjo Giarratana`, and the title is `Senior Staff`, we get output from the query. We don't recommend writing queries like this—use a regular `WHERE` clause with multiple `AND` conditions instead—but it does illustrate exactly what's going on. For an exercise, try writing this query using a join.

Row subqueries require that the number, order, and type of values in the columns match. So, for example, our previous example matches an `INT` to an `INT`, and two character strings to two character strings.

The EXISTS and NOT EXISTS Clauses

You've now seen three types of subquery: scalar subqueries, column subqueries, and row subqueries. In this section, you'll learn about a fourth type, the *correlated subquery*, where a table used in the outer query is referenced in the subquery. Correlated subqueries are often used with the `IN` statement we've already discussed and almost always used with the `EXISTS` and `NOT EXISTS` clauses that are the focus of this section.

EXISTS and NOT EXISTS basics

Before we start on our discussion of correlated subqueries, let's investigate what the `EXISTS` clause does. We'll need a simple but strange example to introduce the clause, since we're not discussing correlated subqueries just yet. So, here goes: suppose you want to find a count of all films in the database, but only if the database is active, which you've defined to mean only if at least one movie from any branch has been rented. Here's the query that does it (don't forget to connect to the `sakila` database again before running this query—hint: use the `use <db>` command):

```
mysql> SELECT COUNT(*) FROM film
    -> WHERE EXISTS (SELECT * FROM rental);
```

```
+----------+
| COUNT(*) |
+----------+
|     1000 |
+----------+
1 row in set (0.01 sec)
```

The subquery returns all rows from the `rental` table. However, what's important is that it returns at least one row; it doesn't matter what's in the row, how many rows there are, or whether the row contains only `NULL` values. So, you can think of the subquery as being true or false, and in this case it's true because it produces some output. When the subquery is true, the outer query that uses the `EXISTS` clause returns a row. The overall result is that all rows in the `film` table are counted because, for each one, the subquery is true.

Let's try a query where the subquery isn't true. Again, let's contrive a query: this time, we'll output the names of all films in the database, but only if a particular film exists. Here’s the query:

```
mysql> SELECT title FROM film
    -> WHERE EXISTS (SELECT * FROM film
    -> WHERE title = 'IS THIS A MOVIE?');
```

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

Since the subquery isn’t true—no rows are returned because `IS THIS A MOVIE?` isn’t in our database—no results are returned by the outer query.

The `NOT EXISTS` clause does the opposite. Imagine you want a list of all actors if you *don’t* have a particular movie in the database. Here it is:

```
mysql> SELECT * FROM actor WHERE NOT EXISTS
    -> (SELECT * FROM film WHERE title = 'ZHIVAGO CORE'
```

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

This time, the inner query is true, but the `NOT EXISTS` clause negates it to give false. Since it’s false, the outer query doesn’t produce results.

You’ll notice that the subquery begins with `SELECT * FROM film`. It doesn’t actually matter what you select in an inner query when you’re using the `EXISTS` clause, since it’s not used by the outer query anyway. You can select one column, everything, or even a constant (as in `SELECT 'cat' from film`), and it’ll have the same effect. Traditionally, though, you’ll see most SQL authors write `SELECT *` by convention.

Correlated subqueries

So far, it’s probably difficult to imagine what you’d do with the `EXISTS` and `NOT EXISTS` clauses. This section shows you how they’re really used, illustrating the most advanced type of nested query that you’ll typically see in action.

Let’s think about the realistic kinds of information you might want from the `sakila` database. Suppose you want a list of all employees who’ve rented something from our company, or are just customers. You can do this easily with a join query, which we recommend you try to think about before you continue. You can also do it with the following nested query that uses a correlated subquery:

```
mysql> SELECT first_name, last_name FROM staff
    -> WHERE EXISTS (SELECT * FROM customer
    -> WHERE customer.first_name = staff.first_name
    -> AND customer.last_name = staff.last_name);
```

```
Empty set (0.01 sec)
```

There’s no output because nobody from the staff is also a customer (or that’s forbidden, but we’ll bend the rules). Let’s add a customer with the same details as one of the staff members:

```
mysql> INSERT INTO customer(store_id, first_name, last_
    -> email, address_id, create_date)
    -> VALUES (1, 'Mike', 'Hillyer',
    -> 'Mike.Hillyer@sakilastaff.com', 3, NOW());
```

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

And try the query again:

```
mysql> SELECT first_name, last_name FROM staff
    -> WHERE EXISTS (SELECT * FROM customer
    -> WHERE customer.first_name = staff.first_name
    -> AND customer.last_name = staff.last_name);
```

```
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| Mike       | Hillyer   |
+------------+-----------+
1 row in set (0.00 sec)
```

So, the query works; now, we just need to understand how!

Let's examine the subquery in our previous example. You can see that it lists only the `customer` table in the `FROM` clause, but it uses a column from the `staff` table in the `WHERE` clause. If you run it in isolation, you'll see this isn't allowed:

```
mysql> SELECT * FROM customer WHERE customer.first_name
```

```
ERROR 1054 (42S22): Unknown column 'staff.first_name' i
```

However, it's legal when executed as a subquery because tables listed in the outer query are allowed to be accessed in the subquery. So, in this example, the current value of `staff.first_name` and `staff.last_name` in the outer query is supplied to the subquery as a constant, scalar value and compared to the customer's first and last names. If the customer's name matches the staff member's name, the subquery is true, and so the outer query outputs a row. Consider two cases that illustrate this more clearly:

- When the `first_name` and `last_name` being processed by the outer query are `Jon` and `Stephens`, the subquery is false because `SELECT * FROM customer WHERE first_name = 'Jon' and last_name = 'Stephens';` doesn't return any rows, and so the staff row for Jon Stephens isn't output as an answer.
- When the `first_name` and `last_name` being processed by the outer query are `Mike` and `Hillyer`, the subquery is true because `SELECT * FROM customer WHERE first_name = 'Mike' and last_name = 'Hillyer';` returns at least one row. Overall, the staff row for Mike Hillyer is returned.

Can you see the power of correlated subqueries? You can use values from the outer query in the inner query to evaluate complex information needs.

We'll now explore another example using `EXISTS`. Let's try to find a count of all films of which we own at least two copies. To do this with `EXISTS`, we need to think through what the inner and outer queries should do. The inner query should produce a result only when the condition we're checking is true; in this case, it should produce output when there are at least two rows in the inventory for the same film. The outer query should increment the counter whenever the inner query is true. Here's the query:

```
mysql> SELECT COUNT(*) FROM film WHERE EXISTS
    -> (SELECT film_id FROM inventory
    -> WHERE inventory.film_id = film.film_id
    -> GROUP BY film_id HAVING COUNT(*) >= 2);
```

```
+----------+
| COUNT(*) |
+----------+
|      958 |
+----------+
1 row in set (0.00 sec)
```

This is yet another query where nesting isn't necessary and a join would suffice, but let's stick with this version for the purpose of explanation. Have a look at the inner query: you can see that the `WHERE` clause ensures that films match by the unique `film_id`, and only matching rows for the current film are considered by the subquery. The `GROUP BY` clause clusters the rows for that film, but only if there are at least two entries in the inventory. Therefore, the inner query produces output only when there are at least two rows for the current film in our inventory. The outer query is straightforward: it can be thought of as incrementing a counter when the subquery produces output.

Here's one more example before we move on and discuss other issues. This example will be in the `employees` database, so switch your client. We've already shown you a query that uses `IN` and finds managers who also had some other position:

```
mysql> SELECT emp_no, first_name, last_name
    -> FROM employees JOIN titles USING (emp_no)
    -> WHERE title = 'Manager'
    -> AND emp_no IN (SELECT emp_no FROM employees
    -> JOIN titles USING (emp_no) WHERE
    -> title <> 'Manager');
```

```
+--------+------------+-------------+
| emp_no | first_name | last_name   |
+--------+------------+-------------+
| 110022 | Margareta  | Markovitch  |
| 110039 | Vishwani   | Minakawa    |
| ...                               |
| 111877 | Xiaobin      | Spinelli     |
| 111939 | Yuchang      | Weedman      |
+--------+-------------+--------------+
24 rows in set (0.11 sec)
```

Let’s rewrite the query to use `EXISTS`. First, think about the subquery: it should produce output when there’s a `title` record for an employee with the same name as a manager.

Second, think about the outer query: it should return the employee’s name when the inner query produces output. Here’s the rewritten query:

```
mysql> SELECT emp_no, first_name, last_name
    -> FROM employees JOIN titles USING (emp_no)
    -> WHERE title = 'Manager'
    -> AND EXISTS (SELECT emp_no FROM titles
    -> WHERE titles.emp_no = employees.emp_no
    -> AND title <> 'Manager');
```

```
+--------+-------------+--------------+
| emp_no | first_name  | last_name    |
+--------+-------------+--------------+
| 110022 | Margareta   | Markovitch   |
| 110039 | Vishwani    | Minakawa     |
| ...                                 |
| 111877 | Xiaobin     | Spinelli     |
| 111939 | Yuchang     | Weedman      |
+--------+-------------+--------------+
24 rows in set (0.09 sec)
```

Again, you can see that the subquery references the `emp_no` column, which comes from the outer query.

Correlated subqueries can be used with any nested query type. Here’s the previous `IN` query rewritten with an outer reference:

```
mysql> SELECT emp_no, first_name, last_name
    -> FROM employees JOIN titles USING (emp_no)
    -> WHERE title = 'Manager'
    -> AND emp_no IN (SELECT emp_no FROM titles
    -> WHERE titles.emp_no = employees.emp_no
    -> AND title <> 'Manager');
```

```
+--------+------------+--------------+
| emp_no | first_name | last_name    |
+--------+------------+--------------+
| 110022 | Margareta  | Markovitch   |
| 110039 | Vishwani   | Minakawa     |
| ...                                |
| 111877 | Xiaobin    | Spinelli     |
| 111939 | Yuchang    | Weedman      |
+--------+------------+--------------+
24 rows in set (0.09 sec)
```

The query is more convoluted than it needs to be, but it illustrates the idea. You can see that the `emp_no` in the subquery references the `employees` table from the outer query.

If the query would return a single row, it can also be rewritten to use an equals instead of `IN` :

```
mysql> SELECT emp_no, first_name, last_name
    -> FROM employees JOIN titles USING (emp_no)
    -> WHERE title = 'Manager'
    -> AND emp_no = (SELECT emp_no FROM titles
    -> WHERE titles.emp_no = employees.emp_no
    -> AND title <> 'Manager');


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

This doesn't work in this case because the subquery returns more than one scalar value. Let's narrow it down:

```
mysql> SELECT emp_no, first_name, last_name
    -> FROM employees JOIN titles USING (emp_no)
    -> WHERE title = 'Manager'
    -> AND emp_no = (SELECT emp_no FROM titles
    -> WHERE titles.emp_no = employees.emp_no
    -> AND title = 'Senior Engineer');


+--------+------------+-----------+
| emp_no | first_name | last_name |
+--------+------------+-----------+
| 110344 | Rosine     | Cools     |
| 110420 | Oscar      | Ghazalie  |
| 110800 | Sanjoy     | Quadeer   |
+--------+------------+-----------+
3 rows in set (0.10 sec)
```

It works now—there's only one manager and senior engineer title with each name—so the column subquery operator `IN` isn't necessary. Of course, if titles are duplicated (for example, if a person switches back and forth between positions), you'd need to use `IN`, `ANY`, or `ALL` instead.

Nested Queries in the FROM Clause

The techniques we've shown all use nested queries in the `WHERE` clause. This section shows you how they can alternatively be used in the `FROM` clause. This is useful when you want to manipulate the source of the data you're using in a query.

In the `employees` database, the `salaries` table stores the annual wage alongside the employee ID. If you want to find the monthly rate, for example, you can do some math in the query. One option in this case is to do it with a subquery:

```
mysql> SELECT emp_no, monthly_salary FROM
    -> (SELECT emp_no, salary/12 AS monthly_salary FROM
    -> LIMIT 5;
```

```
+--------+----------------+
| emp_no | monthly_salary |
+--------+----------------+
|  10001 |      5009.7500 |
|  10001 |      5175.1667 |
|  10001 |      5506.1667 |
|  10001 |      5549.6667 |
|  10001 |      5580.0833 |
+--------+----------------+
5 rows in set (0.00 sec)
```

Focus on what follows the `FROM` clause. The subquery uses the `salaries` table and returns two columns: the first column is the `emp_no`; the second column is aliased as `monthly_salary` and is the `salary` value divided by 12. The outer query is straightforward: it just returns the `emp_no` and the `monthly_salary` value created through the subquery. Note that we've added the table alias `ms` for the subquery. When we use a subquery as a table —that is, we use a `SELECT FROM` operation on it—this "derived table" must have an alias, even if we don't use the alias in our query. MySQL complains if we omit the alias:

```
mysql> SELECT emp_no, monthly_salary FROM
    -> (SELECT emp_no, salary/12 AS monthly_salary FROM
    -> LIMIT 5;
```

```
ERROR 1248 (42000): Every derived table must have its c
```

Here's another example, now in the `sakila` database. Suppose we want to find out the average sum a film brings us through rentals, or the average gross, as we'll call it. Let's begin by thinking through the subquery. It should return the sum of payments that we have for each film. Then, the outer query should average the values to give the answer. Here's the query:

```
mysql> SELECT AVG(gross) FROM
    -> (SELECT SUM(amount) AS gross
    -> FROM payment JOIN rental USING (rental_id)
    -> JOIN inventory USING (inventory_id)
    -> JOIN film USING (film_id)
    -> GROUP BY film_id) AS gross_amount;
```

```
+------------+
| AVG(gross) |
+------------+
|  70.361754 |
+------------+
1 row in set (0.05 sec)
```

You can see that the inner query joins together `payment`, `rental`, `inventory`, and `film`, and groups the sales together by film so you can get a sum for each film. If you run it in isolation, here's what happens:

```
mysql> SELECT SUM(amount) AS gross
    -> FROM payment JOIN rental USING (rental_id)
    -> JOIN inventory USING (inventory_id)
    -> JOIN film USING (film_id)
    -> GROUP BY film_id;
```

```
+--------+
| gross  |
+--------+
|  36.77 |
|  52.93 |
|  37.88 |
|    ... |
|  14.91 |
|  73.83 |
| 214.69 |
+--------+
958 rows in set (0.08 sec)
```

Now, the outer query takes these sums—which are aliased as `gross` --and averages them to give the final result. This query is the typical way that you apply two aggregate functions to one set of data. You can't apply aggregate functions in a cascade, as in `AVG(SUM(amount))` :

```
mysql> SELECT AVG(SUM(amount)) AS avg_gross
    -> FROM payment JOIN rental USING (rental_id)
    -> JOIN inventory USING (inventory_id)
    -> JOIN film USING (film_id) GROUP BY film_id;
```

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

With subqueries in `FROM` clauses, you can return a scalar value, a set of column values, more than one row, or even a whole table. However, you can't use correlated subqueries, meaning that you can't reference tables or columns from tables that aren't explicitly listed in the subquery. Note also that you must alias the whole subquery using the `AS` keyword and give it a name, even if you don't use that name anywhere in the query.

Nested Queries in JOINs

The last use of nested queries we'll show, but not the least useful, is using them in joins. In this use case, the results of the subquery basically form a new table and can be used in any of the join types we have discussed.

For an example of this, let's go back to the query that listed the number of films from each of the categories a particular customer has rented. Remember, we had an issue writing that query using just joins: we didn't get a zero count for categories from which our customer didn't rent. This was the query:

```
mysql> SELECT cat.name AS category_name, COUNT(cat.cate
    -> FROM category AS cat LEFT JOIN film_category USI
    -> LEFT JOIN inventory USING (film_id)
    -> LEFT JOIN rental USING (inventory_id)
    -> JOIN customer AS cs ON rental.customer_id = cs.c
    -> WHERE cs.email = 'WESLEY.BULL@sakilacustomer.org
    -> GROUP BY category_name ORDER BY cnt DESC;
```

```
+-------------+-----+
| name        | cnt |
+-------------+-----+
| Games       |   9 |
| Foreign     |   6 |
| ...               |
| ...               |
| Comedy      |   1 |
| Sports      |   1 |
+-------------+-----+
14 rows in set (0.00 sec)
```

Now that we know about subqueries and joins and that subqueries can be used in joins, we can easily finish the task. This is our new query:

```
mysql> SELECT cat.name AS category_name, cnt
    -> FROM category AS cat
    -> LEFT JOIN (SELECT cat.name, COUNT(cat.category_i
    ->    FROM category AS cat
    ->    LEFT JOIN film_category USING (category_id)
    ->    LEFT JOIN inventory USING (film_id)
    ->    LEFT JOIN rental USING (inventory_id)
    ->    JOIN customer cs ON rental.customer_id = cs.c
    ->    WHERE cs.email = 'WESLEY.BULL@sakilacustomer.
    ->    GROUP BY cat.name) customer_cat USING (name)
    -> ORDER BY cnt DESC;
```

```
+-------------+------+
| name        | cnt  |
+-------------+------+
| Games       |    9 |
| Foreign     |    6 |
| ...                |
| Children    |    1 |
| Sports      |    1 |
| Sci-Fi      | NULL |
| Action      | NULL |
| Thriller    | NULL |
+-------------+------+
17 rows in set (0.01 sec)
```

Finally, we get all the categories displayed, and we get `NULL` values for those where no rentals were made. Let's review what's going on in our new query. The subquery, which we aliased as `customer_cat`, is our previous query without the `ORDER BY` clause. Thus, we know what it will return: 14 rows for categories in which Wesley rented something, and the number of rentals in each. Next, use `LEFT JOIN` to concatenate that information to the full list of categories from the `category` table. The `category` table is driving the join, so it'll have every row selected. We join the subquery using the `name` column that matches between the subquery's output and the `category` table's column.

The technique we showed here is a very powerful one; however, as always with subqueries, it comes at a cost. MySQL cannot optimize the whole query as efficiently when a subquery is present in the join clause.

User Variables

Often you'll want to save values that are returned from queries. You might want to do this so that you can easily use a value in a later query. You might also simply want to save a result for later display. In both cases, user variables solve the problem: they allow you to store a result and use it later.

Let's illustrate user variables with a simple example. The following query finds the title of a film and saves the result in a user variable:

```
mysql> SELECT @film:=title FROM film WHERE film_id = 1;
```

```
+------------------+
| @film:=title     |
+------------------+
| ACADEMY DINOSAUR |
+------------------+
1 row in set, 1 warning (0.00 sec)
```

The user variable is named `film`, and it's denoted as a user variable by the `@` character that precedes it. The value is assigned using the `:=` operator. You can print out the contents of the user variable with the following very short query:

```
mysql> SELECT @film;
```

```
+------------------+
| @film            |
+------------------+
| ACADEMY DINOSAUR |
+------------------+
1 row in set (0.00 sec)
```

You may have noticed the warning—what was that about?

```
mysql> SELECT @film:=title FROM film WHERE film_id = 1;
mysql> SHOW WARNINGS\G
```

```
*************************** 1. row ********************
  Level: Warning
   Code: 1287
Message: Setting user variables within expressions is d
```

```
and will be removed in a future release. Consider alter
'SET variable=expression, ...', or
'SELECT expression(s) INTO variables(s)'.
1 row in set (0.00 sec)
```

Let's cover the two alternatives proposed. First, we can still execute a nested query within a `SET` statement:

```
mysql> SET @film := (SELECT title FROM film WHERE film_
```

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

```
mysql> SELECT @film;
```

```
+------------------+
| @film            |
+------------------+
| ACADEMY DINOSAUR |
+------------------+
1 row in set (0.00 sec)
```

Second, we can use the `SELECT INTO` statement:

```
mysql> SELECT title INTO @film FROM film WHERE film_id
```

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

```
mysql> SELECT @film;
```

```
+------------------+
| @film            |
+------------------+
| ACADEMY DINOSAUR |
+------------------+
1 row in set (0.00 sec)
```

You can explicitly set a variable using the `SET` statement without a `SELECT`. Suppose you want to initialize a counter to zero:

```
mysql> SET @counter := 0;

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

The `:=` is optional, and you can write `=` instead and mix them up. You should separate several assignments with a comma or put each in a statement of its own:

```
mysql> SET @counter = 0, @age := 23;

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

The alternative syntax for `SET` is `SELECT INTO`. You can initialize a single variable:

```
mysql> SELECT 0 INTO @counter;

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

Or multiple variables at once:

```
mysql> SELECT 0, 23 INTO @counter, @age;

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

The most common use of user variables is to save a result and use it later. You'll recall the following example from earlier in the chapter, which we used to motivate nested queries (which are certainly a better solution for this problem). Here, we want to find the name of the film that a particular customer rented most recently:

```
mysql> SELECT MAX(rental_date) FROM rental
    -> JOIN customer USING (customer_id)
    -> WHERE email = 'WESLEY.BULL@sakilacustomer.org';
```

```
+---------------------+
| MAX(rental_date)    |
+---------------------+
| 2005-08-23 15:46:33 |
+---------------------+
1 row in set (0.01 sec)
```

```
mysql> SELECT title FROM film
    -> JOIN inventory USING (film_id)
    -> JOIN rental USING (inventory_id)
    -> JOIN customer USING (customer_id)
    -> WHERE email = 'WESLEY.BULL@sakilacustomer.org'
    -> AND rental_date = '2005-08-23 15:46:33';
```

```
+-------------+
| title       |
+-------------+
| KARATE MOON |
+-------------+
1 row in set (0.00 sec)
```

You can use a user variable to save the result for input into the following query. Here's the same query pair rewritten using this approach:

```
mysql> SELECT MAX(rental_date) INTO @recent FROM rental
    -> JOIN customer USING (customer_id)
    -> WHERE email = 'WESLEY.BULL@sakilacustomer.org';
```

```
1 row in set (0.01 sec)
```

```
mysql> SELECT title FROM film
    -> JOIN inventory USING (film_id)
    -> JOIN rental USING (inventory_id)
    -> JOIN customer USING (customer_id)
    -> WHERE email = 'WESLEY.BULL@sakilacustomer.org'
    -> AND rental_date = @recent;
```

```
+-------------+
| title       |
+-------------+
| KARATE MOON |
+-------------+
1 row in set (0.00 sec)
```

This can save you cutting and pasting, and it certainly helps you avoid typing errors.

Here are some guidelines on using user variables:

- User variables are unique to a connection: variables that you create can't be seen by anyone else, and two different connections can have two different variables with the same name.
- The variable names can be alphanumeric strings and can also include the period (`.`), underscore (`_`), and dollar sign (`$`) characters.
- Variable names are case-sensitive in MySQL versions earlier than version 5, and case-insensitive from version 5 onward.
- Any variable that isn't initialized has the value `NULL` ; you can also manually set a variable to be `NULL` .
- Variables are destroyed when a connection closes.
- You should avoid trying to both assign a value to a variable and use the variable as part of a `SELECT` query. Two reasons for this are that the new value may not be available for use immediately in the same statement, and a variable's type is set when it's first assigned in a query; trying to use it later as a different type in the same SQL statement can lead to unexpected results.
  Let's look at the first issue in more detail using the new variable `@fid` . Since we haven't used this variable before, it's empty. Now, let's show the `film_id` for movies that have an entry in the `inventory` table. Instead of showing it directly, we'll assign the `film_id` to the `@fid` variable. Our query will show the variable three times—once before the assignment operation, once as part of the assignment operation, and once afterward:

```
mysql> SELECT @fid, @fid:=film.film_id, @fid FROM fi
    -> WHERE inventory.film_id = @fid;
```

```
Empty set, 1 warning (0.16 sec)
```

This returns nothing apart from a deprecation warning; since there's nothing in the variable to start with, the `WHERE` clause tries to look for empty `inventory.film_id` values. If we modify the query to use `film.film_id` as part of the `WHERE` clause, things work as expected:

```
mysql> SELECT @fid, @fid:=film.film_id, @fid FROM fi
    -> WHERE inventory.film_id = film.film_id LIMIT
```

```
+------+--------------------+------+
| @fid | @fid:=film.film_id | @fid |
+------+--------------------+------+
| NULL |                  1 | 1    |
| 1    |                  1 | 1    |
| 1    |                  1 | 1    |
| ...                              |
| 4    |                  4 | 4    |
| 4    |                  4 | 4    |
+------+--------------------+------+
20 rows in set, 1 warning (0.00 sec)
```

Now that if `@fid` isn't empty, the initial query will produce some results:

```
mysql> SELECT @fid, @fid:=film.film_id, @fid FROM fi
    -> WHERE inventory.film_id = film.film_id LIMIT
```

```
+------+--------------------+------+
| @fid | @fid:=film.film_id | @fid |
+------+--------------------+------+
|    4 |                  1 |    1 |
|    1 |                  1 |    1 |
|  ...                             |
|    4 |                  4 |    4 |
|    4 |                  4 |    4 |
+------+--------------------+------+
20 rows in set, 1 warning (0.00 sec)
```

It’s best to avoid such circumstances where the behavior is not guaranteed and is hence unpredictable.