5

JOINING DATABASE TABLES

A SQL query walks into a bar, approaches two tables, and asks, "May I join you?"
—The worst database joke in history

Now that you've learned how to use SQL to select and filter data from a table, you'll see how to join database tables. *Joining* tables means selecting data from more than one table and combining it in a single result set. MySQL provides syntax to do different types of joins, like inner joins and outer joins. In this chapter, you'll look at how to use each type.

Selecting Data from Multiple Tables

The data you want to retrieve from a database often will be stored in more than one table, so you need to return it as one dataset in order to view all of it at once.

Let's look at an example. This table, called `subway_system`, contains data for every subway in the world:

```
subway_system              city              country_code
-------------------------  ----------------  ------------
Buenos Aires Underground   Buenos Aires      AR
Sydney Metro               Sydney            AU
Vienna U-Bahn              Vienna            AT
Montreal Metro             Montreal          CA
Shanghai Metro             Shanghai          CN
London Underground         London            GB
MBTA                       Boston            US
Chicago L                  Chicago           US
BART                       San Francisco     US
Washington Metro           Washington, D.C.  US
Caracas Metro              Caracas           VE
--snip--
```

The first two columns, `subway_system` and `city`, contain the name of the subway and the city where it's located. The third column, `country_code`, stores the two-character ISO country code. `AR` stands for Argentina, `CN` stands for China, and so on.

This second table, called `country`, has two columns, `country_code` and `country`:

```
country_code   country
------------   -----------
AR             Argentina
AT             Austria
AU             Australia
BD             Bangladesh
BE             Belgium
--snip--
```

Say you want to get a list of subway systems and their full city and country names. That data is spread across the two tables, so you'll need to join them to get the result set you want. Each table has the same `country_code` column, so you'll use that as a link to write a SQL query that joins the tables (see Listing 5-1).

```
select subway_system.subway_system,
       subway_system.city,
       country.country
from   subway_system
inner join country
on     subway_system.country_code = country.country_code;
```

Listing 5-1: Joining the `subway_system` and `country` tables

In the `country` table, the `country_code` column is the primary key. In the `subway_system` table, the `country_code` column is a foreign key. Recall that a primary key uniquely identifies rows in a table, and a foreign key is used to join with the primary key of another table. You use the `=` (equal) symbol

to specify that you want to join all equal values from the `subway_system` and `country` tables' `country_code` columns.

Since you're selecting from two tables in this query, it's a good idea to specify which table the column is in every time you reference it, especially because the same column appears in both tables. There are two reasons for this. First, it will make the SQL easier to maintain because it will be immediately apparent in the SQL query which columns come from which tables. Second, because both tables have a column named `country_code`, if you don't specify the table name, MySQL won't know which column you want to use and will give an error message. To avoid this, in your `select` statement, type the table name, a period, and then the column name. For example, in Listing 5-1, `subway_system.city` refers to the `city` column in the `subway_system` table.

When you run this query, it returns all of the subway systems with the country names retrieved from the `country` table:

```
subway_system              city              country
-------------------------  ----------------  -------------
Buenos Aires Underground   Buenos Aires      Argentina
Sydney Metro               Sydney            Australia
Vienna U-Bahn              Vienna            Austria
Montreal Metro             Montreal          Canada
Shanghai Metro             Shanghai          China
London Underground         London            United Kingdom
MBTA                       Boston            United States
Chicago L                  Chicago           United States
BART                       San Francisco     United States
Washington Metro           Washington, D.C.  United States
Caracas Metro              Caracas           Venezuela
--snip--
```

Note that the `country_code` column does not appear in the resulting join. This is because you selected only the `subway_system`, `city`, and `country` columns in the query.

NOTE *When joining two tables based on columns with the same name, you can use the `using` keyword instead of `on`. For example, replacing the last line in Listing 5-1 with `using (country_code);` would return the same result with less typing required.*

Table Aliasing

To save time when writing SQL, you can declare aliases for your table names. A *table alias* is a short, temporary name for a table. The following query returns the same result set as Listing 5-1:

```
select  s.subway_system,
        s.city,
        c.country
from    subway_system s
inner join country c
on      s.country_code = c.country_code;
```

You declare `s` as the alias for the `subway_system` table and `c` for the `country` table. Then you can type `s` or `c` instead of the full table name when referencing the column names elsewhere in the query. Keep in mind that table aliases are only in effect for the current query.

You can also use the word `as` to define table aliases:

```
select  s.subway_system,
        s.city,
        c.country
from    subway_system as s
inner join country as c
on      s.country_code = c.country_code;
```

The query returns the same results with or without `as`, but you'll cut down on typing by not using it.

Types of Joins

MySQL has several different types of joins, each of which has its own syntax, as summarized in Table 5-1.

Table 5-1: MySQL Join Types

Join type	Description	Syntax
Inner join	Returns rows where both tables have a matching value.	`inner join` `join`

Outer join	Returns all rows from one table and the matching rows from a second table. Left joins return all rows from the table on the left. Right joins return all rows from the table on the right.	`left outer join`
		`left join`
		`right outer join`
		`right join`

Natural join	Returns rows based on column names that are the same in both tables.	`natural join`
Cross join	Matches all rows in one table to all rows in another table and returns a Cartesian product.	`cross join`

Let's look at each type of join in more depth.

Inner Joins

Inner joins are the most commonly used type of join. In an inner join, there must be a match in both tables for data to be retrieved.

You performed an inner join on the `subway_system` and `country` tables in Listing 5-1. The returned list had no rows for Bangladesh and Belgium. These countries are not in the `subway_system` table, as they don't have subways; thus, there was not a match in both tables.

Note that when you specify `inner join` in a query, the word `inner` is optional because this is the default join type. The following query performs an inner join and produces the same results as Listing 5-1:

```
select  s.subway_system,
        s.city,
```

```
        c.country
from    subway_system s
join    country c
on      s.country_code = c.country_code;
```

You'll come across MySQL queries that use `inner join` and others that use `join`. If you have an existing codebase or written standards, it's best to follow the practices outlined there. If not, I recommend including the word `inner` for clarity.

Outer Joins

An outer join displays all rows from one table and any matching rows in a second table. In Listing 5-2, you select all countries and display subway systems for the countries if there are any.

```
select  c.country,
        s.city,
        s.subway_system
from    subway_system s right outer join country c
on      s.country_code = c.country_code;
```

Listing 5-2: Performing a right outer join

In this query, the `subway_system` table is considered the left table because it is to the left of the `outer join` syntax, while the `country` table is the right table. Because this is a *right* outer join, this query returns all the rows from the `country` table even if there is no match in the `subway_system` table. Therefore, all the countries appear in the result set, whether or not they have subway systems:

```
country               city          subway_system
--------------------  ------------  ------------------------
United Arab Emirates  Dubai         Dubai Metro
Afghanistan           null          null
Albania               null          null
Armenia               Yerevan       Yerevan Metro
Angola                null          null
Antarctica            null          null
Argentina             Buenos Aires  Buenos Aires Underground
--snip--
```

For countries without matching rows in the `subway_system` table, the `city` and `subway_system` columns display null values.

As with inner joins, the word `outer` is optional; using `left join` and `right join` will produce the same results as their longer equivalents.

The following outer join returns the same results as Listing 5-2, but uses the `left outer join` syntax instead:

```
select  c.country,
        s.city,
        s.subway_system
```

```
from    country c left outer join subway_system s
on      s.country_code = c.country_code;
```

In this query, the order of the tables is switched from Listing 5-2. The `subway_system` table is now listed last, making it the right table. The syntax `country c left outer join subway_system s` is equivalent to `subway_system s right outer join country c` in Listing 5-2. It doesn't matter which join you use as long as you list the tables in the correct order.

Natural Joins

A natural join in MySQL automatically joins tables when they have a column with the same name. Here is the syntax to automatically join two tables based on a column that is found in both:

```
select  *
from    subway_system s
natural join country c;
```

With natural joins, you avoid a lot of the extra syntax required for an inner join. In Listing 5-2, you had to include `on s.country_code = c.country_code` to join the tables based on their common `country_code` column, but with a natural join, you get that for free. The results of this query are as follows:

```
country_code    subway_system              city            country
------------    ------------------------   ------------    --------------
AR              Buenos Aires Underground   Buenos Aires    Argentina
AU              Sydney Metro               Sydney          Australia
AT              Vienna U-Bahn              Vienna          Austria
CA              Montreal Metro             Montreal        Canada
CN              Shanghai Metro             Shanghai        China
GB              London Underground         London          United Kingdom
US              MBTA                       Boston          United States
US              Chicago L                  Chicago         United States
US              BART                       San Francisco   United States
US              Washington Metro           Washington, D.C. United States
VE              Caracas Metro              Caracas         Venezuela
--snip--
```

Notice that you selected all columns from the tables using the `select *` wildcard. Also, although both tables have a `country_code` column, MySQL's natural join was smart enough to display that column just once in the result set.

Cross Joins

MySQL's cross join syntax can be used to get the Cartesian product of two tables. A *Cartesian product* is a listing of every row in one table matched with every row in a second table. For example, say a restaurant has two database tables called `main_dish` and `side_dish`. Each table has three rows and one column.

The `main_dish` table is as follows:

```
main_item
---------
steak
chicken
ham
```

And the `side_dish` table looks like:

```
side_item
----------
french fries
rice
potato chips
```

A Cartesian product of these tables would be a list of all the possible combinations of main dishes and side dishes, and is retrieved using the `cross join` syntax:

```
select     m.main_item,
           s.side_item
from       main_dish m
cross join side_dish s;
```

This query, unlike the others you've seen, doesn't join tables based on columns. There are no primary keys or foreign keys being used. Here are the results of this query:

```
main_item   side_item
---------   ----------
ham         french fries
chicken     french fries
steak       french fries
ham         rice
chicken     rice
steak       rice
ham         potato chips
chicken     potato chips
steak       potato chips
```

Since there are three rows in the `main_dish` table and three rows in the `side_dish` table, the total number of possible combinations is nine.

Self Joins

Sometimes, it can be beneficial to join a table to itself, which is known as a self join. Rather than using special syntax as you did in the previous joins, you perform a self join by listing the same table name twice and using two different table aliases.

For example, the following table, called `music_preference`, lists music fans and their favorite genre of music:

```
music_fan   favorite_genre
---------   --------------
Bob         Reggae
Earl        Bluegrass
Ella        Jazz
Peter       Reggae
Benny       Jazz
Bunny       Reggae
Sierra      Bluegrass
Billie      Jazz
```

To pair music fans who like the same genre, you join the `music_preference` table to itself, as shown in Listing 5-3.

```
Select a.music_fan,
       b.music_fan
from   music_preference a
inner join music_preference b
on (a.favorite_genre = b.favorite_genre)
where  a.music_fan != b.music_fan
order by a.music_fan;
```

Listing 5-3: Self join of the `music_preference` table

The `music_preference` table is listed twice in the query, aliased once as table `a` and once as table `b`. MySQL will then join tables `a` and `b` as if they are different tables.

In this query, you use the `!=` (not equal) syntax in the `where` clause to ensure that the value of the `music_fan` column from table `a` is not the same as the value of the `music_fan` column in table `b`. (Remember from Chapter 3 that you can use a `where` clause in your `select` statements to filter your results by applying certain conditions.) This way, music fans won't be paired up with themselves.

NOTE *The `!=` (not equal) syntax used here and the `=` (equal) syntax you've been using throughout this chapter are what's known as* comparison operators, *as they let you compare values in your MySQL queries. Chapter 7 will discuss comparison operators in more detail.*

Listing 5-3 produces the following result set:

```
music_fan  music_fan
---------  ---------
Benny      Ella
Benny      Billie
Billie     Ella
Billie     Benny
Bob        Peter
Bob        Bunny
```

```
Bunny       Bob
Bunny       Peter
Earl        Sierra
Ella        Benny
Ella        Billie
Peter       Bob
Peter       Bunny
Sierra      Earl
```

A music fan can now find other fans of their favorite genre in the right column next to their name.

NOTE *In Listing 5-3, the table is joined to itself as an inner join, but you could have used another type of join, like an outer join or a cross join.*

Variations on Join Syntax

MySQL allows you to write SQL queries that accomplish the same results in different ways. It's a good idea to get comfortable with different syntaxes, as you may have to modify code created by someone who doesn't write SQL queries in quite the same way that you do.

Parentheses

You can choose to use parentheses when joining on columns or leave them off. This query, which does not use parentheses

```
select  s.subway_system,
        s.city,
        c.country
from    subway_system as s
inner join country as c
on      s.country_code = c.country_code;
```

is the same as this query, which does:

```
select  s.subway_system,
        s.city,
        c.country
from    subway_system as s
inner join country as c
on      (s.country_code = c.country_code);
```

Both queries return the same result.

Old-School Inner Joins

This query, written in an older style of SQL, is equivalent to Listing 5-1:

```
select  s.subway_system,
        s.city,
        c.country
```

```
from    subway_system as s,
        country as c
where   s.country_code = c.country_code;
```

This code doesn't include the word `join`; instead, it lists the table names separated by a comma in the `from` statement.

When writing queries, use the newer syntax shown in Listing 5-1, but keep in mind that this older style is still supported by MySQL and you might see it used in some legacy code today.

Column Aliasing

You read earlier in the chapter about table aliasing; now you'll create aliases for columns.

In some parts of the world, like France, subway systems are referred to as *metros*. Let's select the subway systems for cities in France from the `subway_system` table and use column aliasing to display the heading `metro` instead:

```
select  s.subway_system as metro,
        s.city,
        c.country
from    subway_system as s
inner join country as c
on      s.country_code = c.country_code
where   c.country_code = 'FR';
```

As with table aliases, you can use the word `as` in your SQL query or you can leave it out. Either way, the results of the query are as follows, now with the `subway_system` column heading changed to `metro`:

```
metro           city        country
-----           --------    -------
Lille Metro     Lille       France
Lyon Metro      Lyon        France
Marseille Metro Marseille   France
Paris Metro     Paris       France
Rennes Metro    Rennes      France
Toulouse Metro  Toulouse    France
```

When creating tables, try to give your column headings descriptive names so that the results of your queries will be meaningful at a glance. In cases where the column names could be clearer, you can use a column alias.

Joining Tables in Different Databases

Sometimes there are tables with the same name in multiple databases, so you need to tell MySQL which database to use. There are a couple of different ways to do this.

In this query, the `use` command (introduced in Chapter 2) tells MySQL to use the specified database for the SQL statements that follow it:

```
use subway;

select * from subway_system;
```

On the first line, the `use` command sets the current database to `subway`. Then, when you select all the rows from the `subway_system` table on the next line, MySQL knows to pull data from the `subway_system` table in the `subway` database.

Here's a second way to specify the database name in your `select` statements:

```
select * from subway.subway_system;
```

In this syntax, the table name is preceded by the database name and a period. The `subway.subway_system` syntax tells MySQL that you want to select from the `subway_system` table in the `subway` database.

Both options produce the same result set:

```
subway_system       city                      country_code
-----------------   -----------------------   ------------
Buenos Aires        Underground Buenos Aires  AR
Sydney Metro        Sydney                    AU
Vienna U-Bahn       Vienna                    AT
Montreal Metro      Montreal                  CA
Shanghai Metro      Shanghai                  CN
London Underground  London                    GB
--snip--
```

Specifying the database and table name allows you to join tables that are in different databases on the same MySQL server, like so:

```
select  s.subway_system,
        s.city,
        c.country
from    subway.subway_system as s
inner join location.country as c
on      s.country_code = c.country_code;
```

This query joins the `country` table in the `location` database with the `subway_system` table in the `subway` database.

Summary

In this chapter, you learned how to select data from two tables and display that data in a single result set using various joins offered by MySQL. In Chapter 6, you'll build on this knowledge by performing even more complex joins involving multiple tables.

TRY IT YOURSELF

In the `solar_system` database, there are two tables: `planet` and `ring`. The `planet` table is as follows:

```
planet_id  planet_name
---------  ---------
    1      Mercury
    2      Venus
    3      Earth
    4      Mars
    5      Jupiter
    6      Saturn
    7      Uranus
    8      Neptune
```

The `ring` table stores only the planets with rings:

```
planet_id   ring_tot
---------   --------
    5           3
    6           7
    7          13
    8           6
```

5-1. Write a SQL query to perform an inner join between the `planet` and the `ring` tables, joining the tables based on their `planet_id` columns. How many rows do you expect the query to return?

5-2. Write a SQL query to do an outer join between the `planet` and the `ring` tables, with the `planet` table as the *left* table.

5-3. Modify your SQL query from Exercise 5-2 so that the `planet` table is the *right* table. The set returned by the query should be the same as the results of the previous exercise.

5-4. Modify your SQL query from Exercise 5-3 using a column alias. Make the `ring_tot` column display as `rings` in the heading of the result set.