Joins in SQL.

Objectives

By the end of this chapter, you should be able to:

- Define **foreign key**
- Create a one to many (1:M) association in SQL
- Create a many to many (M:N) association in SQL

Associations Intro

So far, all of our sql table examples have used a single table. However, one of the most important features of relational databases is the ability to associate (relate) one table to another. The first thing we need to understand in order to create these associations of data is a **foreign key constraint**.

Foreign Key

A foreign key is a constraint placed on a column. It associates the data in one column to a column in a different table. Let's look at an example:

```
CREATE TABLE people (id SERIAL PRIMARY KEY,
                    first_name TEXT,
                    last_name TEXT);

CREATE TABLE interests (id SERIAL PRIMARY KEY,
                       interest TEXT,
                       people_id INTEGER REFERENCES people);
```

In our table definition, we have a **people** table and we have an **interests** table. Using the foreign key constraint in the interest table we have associated the `people_id` column with the `id` column from the `people` table.

Let's look at the two tables visually with some sample data:

people

id	first_name	last_name
1	Billy Jean	King
2	Dawn	Riley
3	Grace	Hopper

interests

id	interest	people_id

id	interest	people_id
1	sailing	2
2	tennis	1
3	software	3
4	debugging	3

In our sample data, the values in the `people_id` column reference the `id` from the people table. Take the row form the `people` table with the `id` of 3 as an example. We can see that id 3, Grace Hopper, is associated with 2 rows in the interests table because the `people_id` matches the id of 3. The two interests are software and debugging.

But why is a foreign key considered a constraint? Well the database actually ensures that the data in your foreign key column exists in the table that the foreign key references. Try inserting into the interests table with data that does not exist in the people table:

```
INSERT INTO interests (interest, people_id) VALUES ('scuba diving', 30);
```

After you run the insert statement, you should see an error like this:

```
ERROR:  insert or update on table "interests" violates foreign key constraint
```

One-to-Many

In our foreign key example above, we actually setup a one-to-many (1:M) association. The way we structured the database implies that one person can have many interests, and that each interest belongs to exactly one person.

One-to-many associations are extremely common in database and they association usually implies that one row from a table owns or has rows from another table.

For example:

```
CREATE TABLE users (id SERIAL PRIMARY KEY,
                    email TEXT,
                    password TEXT);


CREATE TABLE photos (id SERIAL PRIMARY KEY,
                     url TEXT,
                     user_id INTEGER REFERENCES people);
```

In the table above, each user in the users table, can have many photos and one photo from the photos table belongs to exactly one user based on the value of `user_id`.

Another example:

```
CREATE TABLE states (id SERIAL PRIMARY KEY,
                     name TEXT);
```

```
CREATE TABLE cities (id SERIAL PRIMARY KEY,
                     name TEXT,
                     state_id INTEGER REFERENCES states);
```

In this example, a state has many cities and a city belongs to a state. So this is a one to many relationship as well.

Many-to-Many

Often your database needs to model a more complex association in which an item from one table can associate to many rows in another table and vice versa. This is a many to many (M:N) relationship.

Some examples:

- A database with products and order. A single product can have many orders placed on it, and a single order can have many products in it.
- School courses and students. A student can be enrolled in many courses, and a course has many students.
- Recipes and ingredients. A recipe can have many ingredients, and ingredients can be used in many recipes.

Many-to-Many in SQL

Let's create an association between students and courses in SQL. In order to make this association work, we will need a third table that maps a foreign key from courses to a foreign key from students. We will call that third table `enrollments` in this case:

```
CREATE TABLE students (id SERIAL PRIMARY KEY,
                       name TEXT);

CREATE TABLE courses (id SERIAL PRIMARY KEY,
                      name TEXT,
                      teacher_name TEXT);

CREATE TABLE enrollments (id SERIAL PRIMARY KEY,
                          student_id INTEGER REFERENCES students,
                          course_id INTEGER REFERENCES courses);
```

In the students and courses example, one student ID can be associated with one course ID by a single row in the `enrollments` table. If that student is in another course, there will be another row in the `enrollments` table that associates the same student ID to another course ID.

Let's look at some sample data for the example:

students

id	first_name	last_name
1	Billy Jean	King

id	first_name	last_name
2	Dawn	Riley
3	Grace	Hopper

courses

id	name	teacher_name
1	Physics	Mrs. Skiles
2	P.E.	Mr. Carty
3	Biology	Mrs. Waldorf
4	Computer Science	Miss. Flurry

enrollments

id	student_id	course_id
1	1	2
2	2	2
3	3	4
4	3	1
5	3	3
6	1	3

So based on this data, Billy Jean King is taking P.E. and Biology, Dawn Riley is only taking P.E. and Grace Hopper is taking Computer Science, Biology, and Physics.

The same foreign key constraints apply to the enrollments table. If a `student_id` does not exist in the `students` table, then the data will not be allowed to be inserted into the `enrollments` table. Likewise, if a `course_id` does not exist in the `courses` table, the data will not be allowed to be inserted into the `enrollments` table.

What is a Join?

A query with a join combines data from two different tables. Typically, the join combines data **based on a matching row**.

For example, if we have a table like the following:

people

id	first_name	last_name
1	Billy Jean	King
2	Dawn	Riley
3	Grace	Hopper

interests

id	interest	people_id
1	sailing	2

id	interest	people_id
2	tennis	1
3	software	3
4	debugging	3

If we were to do a join on the `id` column from the `people` table with the `people_id` column from the `interests` table, our results would be:

people joined with **interests** on `people.id` = `interests.people_id`:

people.id	first_name	last_name	interests.id	interest	people_id
1	Billy Jean	King	2	tennis	1
2	Dawn	Riley	1	sailing	2
3	Grace	Hopper	3	software	3
3	Grace	Hopper	4	debugging	3

Notice that we get two rows with the person Grace Hopper because in this data set, Grace Hopper has two matches in the `interests` table, therefore, the match is represented with two different rows in the resulting join.

Joins in SQL

Let's start by creating a database and filling it with our sample data. Make sure to delete the database and recreate it if you have made it already. It is important that the `id`s from people start counting from 1.

```
CREATE TABLE people (id SERIAL PRIMARY KEY,
                     first_name TEXT,
                     last_name TEXT);

CREATE TABLE interests (id SERIAL PRIMARY KEY,
                        interest TEXT,
                        people_id INTEGER REFERENCES people);

INSERT INTO people (first_name, last_name) VALUES ('Billy Jean', 'King'),
                                                  ('Dawn', 'Riley'),
                                                  ('Grace', 'Hopper');

INSERT INTO interests (interest, people_id) VALUES ('sailing', 2),
                                                   ('tennis', 1),
                                                   ('software', 3),
                                                   ('debugging', 3);
```

Now that we have some data, let's do our first join:

```
SELECT * FROM people
    INNER JOIN interests ON people.id=interests.people_id;
```

Your output for the query should be the same as our example tables from above. So what are we doing? We are combining all rows from people with all possible rows from interests where the `people.id` is equals to the `interests.people_id`. In our example, the `INNER` part of our join is not necessary. You could also write the join this way:

```
SELECT * FROM people
    JOIN interests ON people.id=interests.people_id;
```

Even thought the `INNER` keyword is not necessary, it helps us remember what kind of join we are doing. Next we'll see other types of joins

Left, Right, Full (Outer) Join

So far we have looked a data set where all data from the first table has at least 1 match in the second table, and vice versa. Sometimes that is not the case though. Consider the following data set (if you are following along, you should drop the previous tables and create a new ones):

```
CREATE TABLE people (id SERIAL PRIMARY KEY,
                     first_name TEXT,
                     last_name TEXT);

CREATE TABLE interests (id SERIAL PRIMARY KEY,
                        interest TEXT,
                        people_id INTEGER REFERENCES people);

INSERT INTO people (first_name, last_name) VALUES ('Billy Jean', 'King'),
                                                  ('Dawn', 'Riley'),
                                                  ('Grace', 'Hopper'),
                                                  ('Moxie', 'Garcia');

INSERT INTO interests (interest, people_id) VALUES ('sailing', 2),
                                                   ('tennis', 1),
                                                   ('software', 3),
                                                   ('debugging', 3),
                                                   ('snow boarding', NULL),
                                                   ('ham radio', NULL);
```

In this data set, Moxie Garcia does not have any interests. Also, none of the people are interested in snow boarding or ham radios.

When we do an inner join, the results are only rows where a person matches an interest. But sometimes we may want to include people without interests, or only find people with no interests. We can solve this problem with left or right joins.

Here is how we would answer the following questions:

Find all people and all of their interests (including people with no interests):

```
SELECT * FROM people
    LEFT JOIN interests ON people.id=interests.people_id;
```

Here are the results:

id	first_name	last_name	id	interest	people_id
2	Dawn	Riley	1	sailing	2
1	Billy Jean	King	2	tennis	1
3	Grace	Hopper	3	software	3
3	Grace	Hopper	4	debugging	3
4	Moxie	Garcia			

The `LEFT JOIN` gets us all the rows from the left (the left in this case is the `people` table) that have a match in the interests table as well as all the people that do not have a match.

We could get the same results by doing a right join. In this case, a right join gets all the matches from the right (the people table) plus all of the people without any interest matches.

```
SELECT * FROM interests
    RIGHT JOIN people ON people.id=interests.people_id;
```

We can also add the keyword `OUTER` to remind ourselves that this is an outer join. Just like `INNER`, this is an optional keyword that you can add, but it doesn't change the query:

```
SELECT * FROM interests
    RIGHT OUTER JOIN people ON people.id=interests.people_id;
```

Find all of the people that have no interests:

```
SELECT * FROM people
    LEFT JOIN interests ON people.id=interests.people_id
    WHERE interests.id IS NULL;
```

In this query, we first get all of the people and all of their interests (including people with no interests), then we limit that result to only the rows without an interest id. In other words, the results where people do not have a matching interest. Your output should be:

id	first_name	last_name	id	interest	people_id
4	Moxie	Garcia			

Find all people and all interests (even if there is no corresponding match)

```
SELECT * FROM people
    FULL JOIN interests ON people.id=interests.people_id;
```

Here are the results:

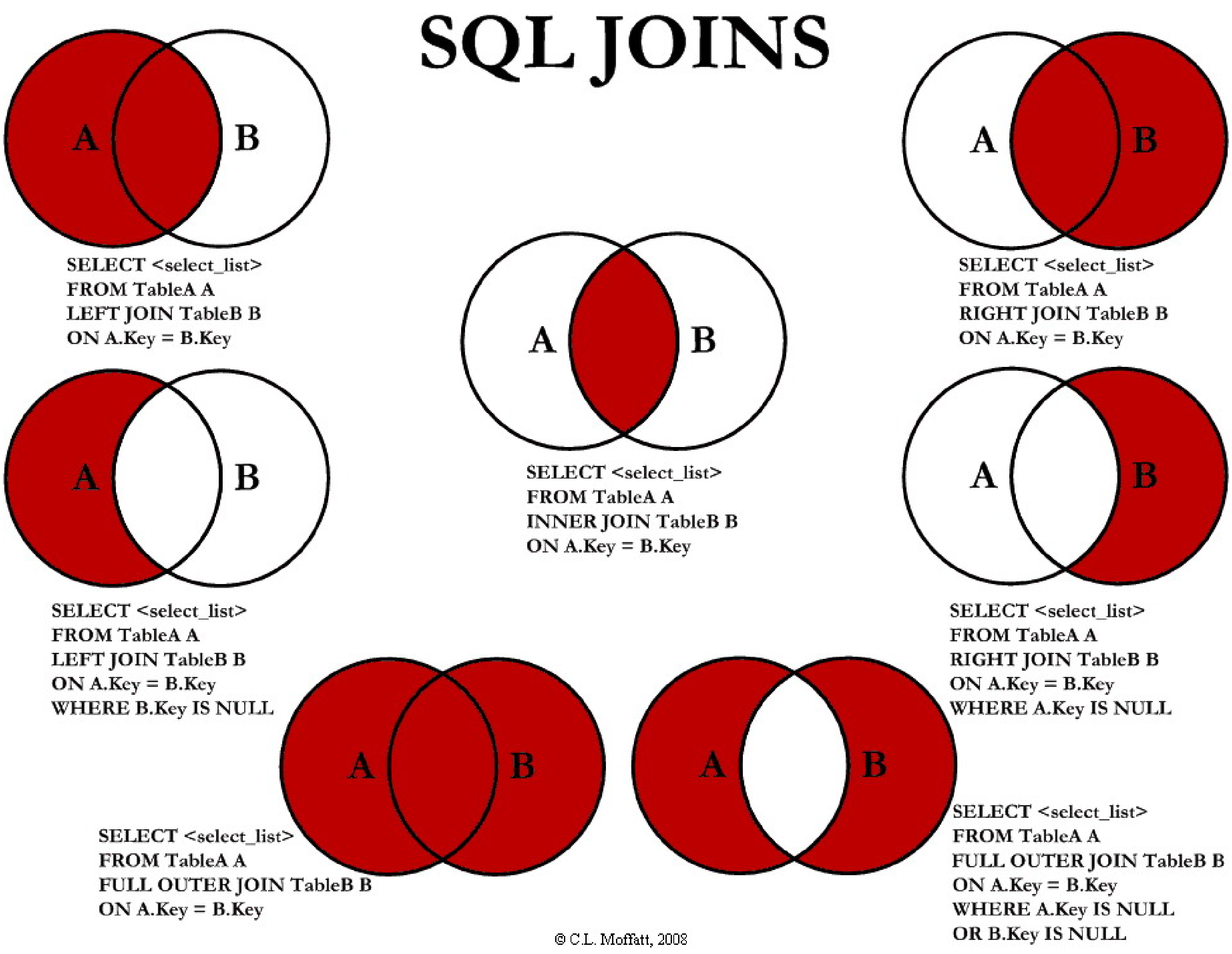
id	first_name	last_name	id	interest	people_id

id	first_name	last_name	id	interest	people_id
2	Dawn	Riley	1	sailing	2
1	Billy Jean	King	2	tennis	1
3	Grace	Hopper	3	software	3
3	Grace	Hopper	4	debugging	3
			5	snow boarding	
			6	ham radio	
4	Moxie	Garcia			

So with our `FULL JOIN` we are getting all matching combinations of people and interests, plus interests that do not have any matching people and people that do not have any matching interests.

Join Diagram

Here is a useful diagram from C.L. Moffet's article that illustrates the common joins which we just covered:



```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```

```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```

```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```

```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```

Cross Join

A cross join is not the most useful join, but it may come in handy once and a while. It gives you all possible combinations of rows from the first table and rows from the second table. Here is the query:

```
SELECT * FROM people
    CROSS JOIN interests;
```

And this is the output:

id	first_name	last_name	id	interest	people_id
1	Billy Jean	King	1	sailing	2
2	Dawn	Riley	1	sailing	2
3	Grace	Hopper	1	sailing	2
4	Moxie	Garcia	1	sailing	2
1	Billy Jean	King	2	tennis	1
2	Dawn	Riley	2	tennis	1
3	Grace	Hopper	2	tennis	1
4	Moxie	Garcia	2	tennis	1
1	Billy Jean	King	3	software	3
2	Dawn	Riley	3	software	3
3	Grace	Hopper	3	software	3
4	Moxie	Garcia	3	software	3
1	Billy Jean	King	4	debugging	3
2	Dawn	Riley	4	debugging	3
3	Grace	Hopper	4	debugging	3
4	Moxie	Garcia	4	debugging	3
1	Billy Jean	King	5	snow boarding	
2	Dawn	Riley	5	snow boarding	
3	Grace	Hopper	5	snow boarding	
4	Moxie	Garcia	5	snow boarding	
1	Billy Jean	King	6	ham radio	
2	Dawn	Riley	6	ham radio	
3	Grace	Hopper	6	ham radio	
4	Moxie	Garcia	6	ham radio	

Self Join

A self join is actually just doing a normal join. The big difference is that the two tables that you are joining are the same. The classic example is an employee table with an ID for a boss. The `boss_id` in the table references another employee in the same table:

employees

id	first_name	last_name	boss_id
1	Billy Jean	King	3
2	Dawn	Riley	3
3	Grace	Hopper	4
4	Ada	Lovelace	
5	Emmy	Noether	4
6	Marie	Curie	5

(In this example data, Billy Jean reports to Grace Hopper, and Ada Lovelace has no boss.)

Here is the sql for the table:

```
CREATE table employees (id SERIAL PRIMARY KEY,
                        first_name TEXT,
                        last_name TEXT,
                        boss_id INTEGER);

INSERT INTO employees (first_name, last_name, boss_id) VALUES ('Billy Jean',
'King', 3),
                                                   ('Dawn', 'Riley', 3),
                                                   ('Grace', 'Hopper', 4),
                                                   ('Ada', 'Lovelace', NULL),
                                                   ('Emmy', 'Noether', 4),
                                                   ('Marie', 'Curie', 5);
```

Find all the employees and their bosses. Also show employees without bosses:

```
SELECT e.id, e.first_name, e.last_name, e.boss_id, b.first_name as
b_first_name, b.last_name as b_last_name
    FROM employees AS e
    LEFT JOIN employees AS b ON e.boss_id=b.id;
```