

# Relationships in SQL

[Download Demo Code <../sql-joins-demo.zip>](#)

## Goals

- Learn what makes SQL databases “relational”
- Understand one-to-many and many-to-many relationships
- Describe and make use of the different types of joins (inner, outer)

## Data Example: Movies

id	title	studio
1	Star Wars: The Force Awakens	Walt Disney Studios Motion Pictures
2	Avatar	20th Century Fox
3	Black Panther	Walt Disney Studios Motion Pictures
4	Jurassic World	Universal Pictures
5	Marvel's The Avengers	Walt Disney Studios Motion Pictures

- So much duplication!
- What if we want other info about studios?

## A Better Way

id	title	studio_id
1	Star Wars: The Force Awakens	1
2	Avatar	2
3	Black Panther	1
4	Jurassic World	3
5	Marvel's The Avengers	1

id	name	founded_in
1	Walt Disney Studios Motion Pictures	1953-06-23
2	20th Century Fox	1935-05-31
3	Universal Pictures	1912-04-30

## One-to-Many (1:M)

- Our **studio\_id** column provides us with a reference to the corresponding record in the **studios** table by its primary key.
- Typically this is implemented with a **foreign key constraint**, which makes sure every **studio\_id** exists somewhere in the **studios** table.
- One-to-Many (1:M) in the sense that one studio *has many* movies, but each movie *has one* studio.
- In this example, we can say **movies** is the *referencing* table, and **studios** is the *referenced* table.

## The Foreign Key Constraint

Setting up a foreign key constraint with DDL:

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

CREATE TABLE movies
(id SERIAL PRIMARY KEY,
 title TEXT,
 studio_id INTEGER REFERENCES studios (id));
```

Constraints are specified by the DDL, but affect DML query behavior.

```
INSERT INTO studios (name, founded_in) VALUES
('Walt Disney Studios Motion Pictures', '1953-06-23'),
('20th Century Fox', '1935-05-31'),
('Universal Pictures', '1912-04-30');
```

```
-- reference Disney's primary key
INSERT INTO movies (title, studio_id)
VALUES ('Star Wars: The Force Awakens', 1);
```

```
-- Throws an Foreign Key Constraint Error...
-- There is no studio with a primary key of 1000
INSERT INTO movies (title, studio_id)
VALUES ('Black Panther', 1000);
```

## Deleting Data Examples

When trying to delete a studio...

We cannot delete it outright while movies still reference it.

```
DELETE FROM studios WHERE id=1; -- error
```

Option 1: Clear out the **studio\_id** columns of movies that reference it.

```
UPDATE movies SET studio_id=NULL WHERE studio_id=1;
DELETE FROM studios WHERE id=1;
```

Option 2: Delete the movies associated with that studio first.

```
DELETE FROM movies WHERE studio_id=1;
DELETE FROM studios WHERE id=1;
```

What are the trade-offs? We will revisit this when we look at how to implement each of the two options above in the DDL.

## Joining Tables

### JOIN Operation

- The **JOIN** operation allows us to create a table in memory by combining information from different tables
- Data from tables is matched according to a join condition
- Most commonly, the join condition involves comparing a **foreign key** from one table and a **primary key** in another table

### Setting Up the Data

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

CREATE TABLE movies
(id SERIAL PRIMARY KEY,
 title TEXT,
 release_year INTEGER,
 runtime INTEGER,
 rating TEXT,
 studio_id INTEGER REFERENCES studios (id));
```

```
INSERT INTO studios
(name, founded_in)
VALUES
('Walt Disney Studios Motion Pictures', '1953-06-23'),
('20th Century Fox', '1935-05-31'),
('Universal Pictures', '1912-04-30');
```

```
INSERT INTO movies
(title, release_year, runtime, rating, studio_id)
VALUES
('Star Wars: The Force Awakens', 2015, 136, 'PG-13', 1),
```

```
('Avatar', 2009, 160, 'PG-13', 2),  
( 'Black Panther', 2018, 140, 'PG-13', 1),  
( 'Jurassic World', 2015, 124, 'PG-13', 3),  
( 'Marvel's The Avengers', 2012, 142, 'PG-13', 1);
```

## Our First Join

```
SELECT title, name  
FROM movies  
JOIN studios  
ON movies.studio_id = studios.id;
```

```
SELECT title, name  
FROM movies  
INNER JOIN studios  
ON movies.studio_id = studios.id;
```

**JOIN** and **INNER JOIN** are the same, the **INNER** keyword is optional.

## Types of Joins

There are two primary types of joins: **inner** and **outer**.

- **Inner**

Only the rows that match the condition in both tables.

- **Outer**

**Left** - All of the rows from the first table (left), combined with matching rows from the second table (right).

**Right** - The matching rows from the first table (left), combined with all the rows from the second table (right).

**Full** - All the rows from both tables (left and right).

## Join Diagrams

### Joins in Practice

- Practically speaking, you'll mostly be using Inner Joins
- Outer joins can be helpful when trying to find rows in one table with no match in another table (e.g. an independent movie with no studio)
- Outer join example:

```
-- this query will include the indie movie  
SELECT name FROM movies
```

```
LEFT JOIN studios
ON movies.studio_id = studios.id;
```

## Many-to-Many

### Movies Revisited

- We've seen an example of a **one-to-many** relationship: one studio has many movies, and one movie belongs to one studio.
- But not every relationship can be expressed in this way...
- Consider actors: one movie has many different actors, but each actor also has roles in many different movies!
- This is an example of a many-to-many relationship.
- A *many-to-many* is just two *one-to-manys* back-to-back!

### Setting Up Actors and Roles

```
-- We've already created the movies database
CREATE TABLE actors
(id SERIAL PRIMARY KEY,
 first_name TEXT,
 last_name TEXT,
 birth_date TEXT);

CREATE TABLE roles
(id SERIAL PRIMARY KEY,
 movie_id INTEGER REFERENCES movies (id),
 actor_id INTEGER REFERENCES actors (id));
```

```
INSERT INTO actors
(first_name, last_name, birth_date)
VALUES
('Scarlett', 'Johansson', '1984-11-22'),
('Samuel L', 'Jackson', '1948-12-21'),
('Kristen', 'Wiig', '1973-08-22');
```

```
INSERT INTO roles
(movie_id, actor_id)
VALUES
(1, 1),
(1, 2),
(3, 2);
```

### Many-to-Many (M:N)

Let's see what the movies, actors and roles tables look like!

id	title	release_year	runtime	rating
1	Marvel's The Avengers	2012	142	PG-13
2	Avatar	2009	160	PG-13
3	Star Wars: Episode I	1999	133	PG

id	first_name	last_name	birth_date
1	Scarlett	Johansson	1984-11-22
2	Samuel L	Jackson	1948-12-21
3	Kristen	Wiig	1973-08-22

id	movie_id	actor_id
1	1	1
2	1	2
3	3	2

## Visualizing the Relationships

Check out [this color-coded spreadsheet](https://docs.google.com/spreadsheets/d/1uFoV781nebAPbtnsQ_qYstib2Mtg99yKVUDXCnXMssE/edit?usp=sharing)

<[https://docs.google.com/spreadsheets/d/1uFoV781nebAPbtnsQ\\_qYstib2Mtg99yKVUDXCnXMssE/edit?usp=sharing](https://docs.google.com/spreadsheets/d/1uFoV781nebAPbtnsQ_qYstib2Mtg99yKVUDXCnXMssE/edit?usp=sharing)>.

## Join Tables

- The **roles** table in our current schema is an example of a join table (aka an associative table aka a mapping table).
- A join table serves as a way to connect two tables in a many-to-many relationship.
- The join table consists of, at a minimum, two foreign key columns to the two other tables in the relationship.
- It is completely valid to put other data in the join table (e.g. how much was an actor paid for the role).
- Sometimes the join table has a nice name (when it has meaning on its own, e.g. **roles**), but you can also just call it **table1\_table2**.

## Querying a Many-to-Many

Connecting movies and actors:

```
SELECT * FROM movies
JOIN roles
ON movies.id = roles.movie_id
JOIN actors
ON roles.actor_id = actors.id;
```

Selecting certain columns, using table alias shorthand:

```
SELECT m.title, a.first_name, a.last_name
FROM movies m
JOIN roles r
  ON m.id = r.movie_id
JOIN actors a
  ON r.actor_id = a.id;
```

Get all the id, first name and last name of the actors that have been in more than one movie

```
SELECT a.id, a.first_name, a.last_name
FROM movies m
JOIN roles r
  ON m.id = r.movie_id
JOIN actors a
  ON r.actor_id = a.id
GROUP BY a.id, a.first_name, a.last_name
HAVING count(*) >= 2;
```

**Your Turn!**