



Funny otter memes

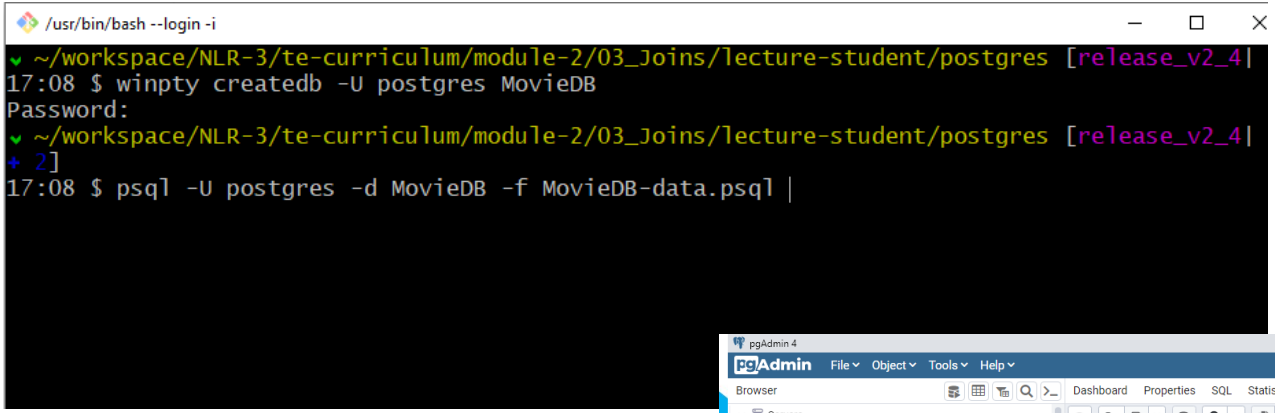
# Module 2-3

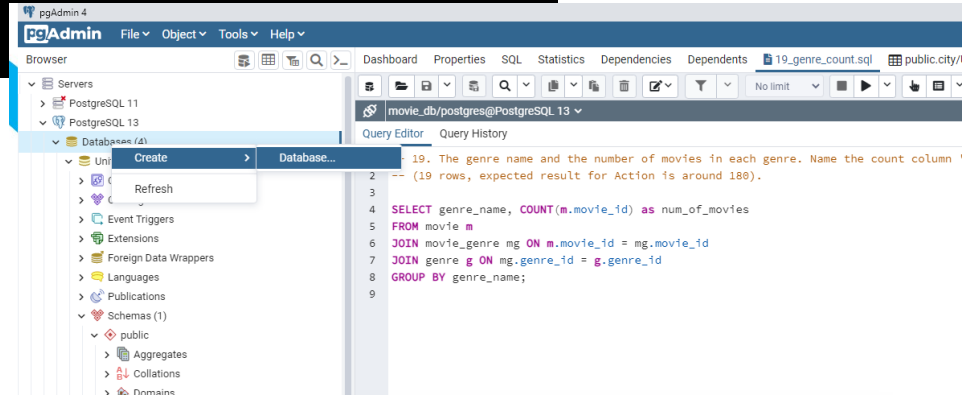
Keys, Joins, Unions

# Objectives

- Keys (Primary, Natural, Surrogate, Foreign)
- Cardinality (1-1, 1-M, M-M)
- SQL Joins (Inner and Left Join)
- Unions

# Create the MovieDB and insert some data



# Amazon Scenario

Users table

*Records for each user*

Shipping_Addresses table

*Shipping Addresses for each user*

Purchases table

*User purchases information*

Products table

*Product Data*

# Keys

In a relational database, all rows must be unique. The column or combination of columns that make it unique are referred to as **key(s)**.

- **Natural Key**: From real world data, SSN's, customer account numbers, driver license numbers
- **Surrogate Key**: Keys artificially created by an application to make a row unique

**Natural Composite Primary Key**

| suit | value | times_played |
|---|---|---|
| Hearts | Ace | 5 |
| Diamonds | Three | 2 |
| Hearts | Jack | 4 |
| Spades | Ace | 1 |

primary key - composite natural key. One column is not enough to identify a unique value, but together they form a unique key

**Surrogate Primary Key**

| id | first_name | last_name |
|---|---|---|
| 1 | Jack | Burton |
| 2 | Gracie | Law |
| 3 | Eddie | Lee |

primary key - surrogate key. There is no value in the data that identifies a unique value, so a unique value is generated for each row.

# Keys

In a relational database, all rows must be unique. The column or combination of columns that make it unique are referred to as **key(s)**.

- **Primary Key**: column or columns in a table that uniquely identify the row. These cannot be duplicated.
  - If you say that SSN is your key, there cannot be more than one row with the same SSN.
- **Foreign Key**: A key that exists in another table, in which the latter is the primary key.

# Foreign Key

A **foreign key** Exist in other tables to reference a unique related row in the source table. Used to create relationships between tables.

- Usually References a **primary key**, but can reference any column that contains a unique value that can be used to identify a specific row.
- Can reference a composite key, but all columns that make up the primary key on the source table must be referenced on the table.

**Country**

| code | name |
|---|---|
| USA | United States |
| GBR | Great Britain |
| CAN | Canada |

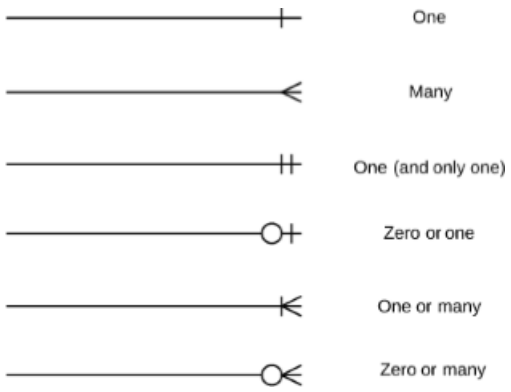
**City**

| id | countrycode | name |
|---|---|---|
| 1 | USA | United States |
| 2 | GBR | Great Britain |
| 3 | CAN | Canada |

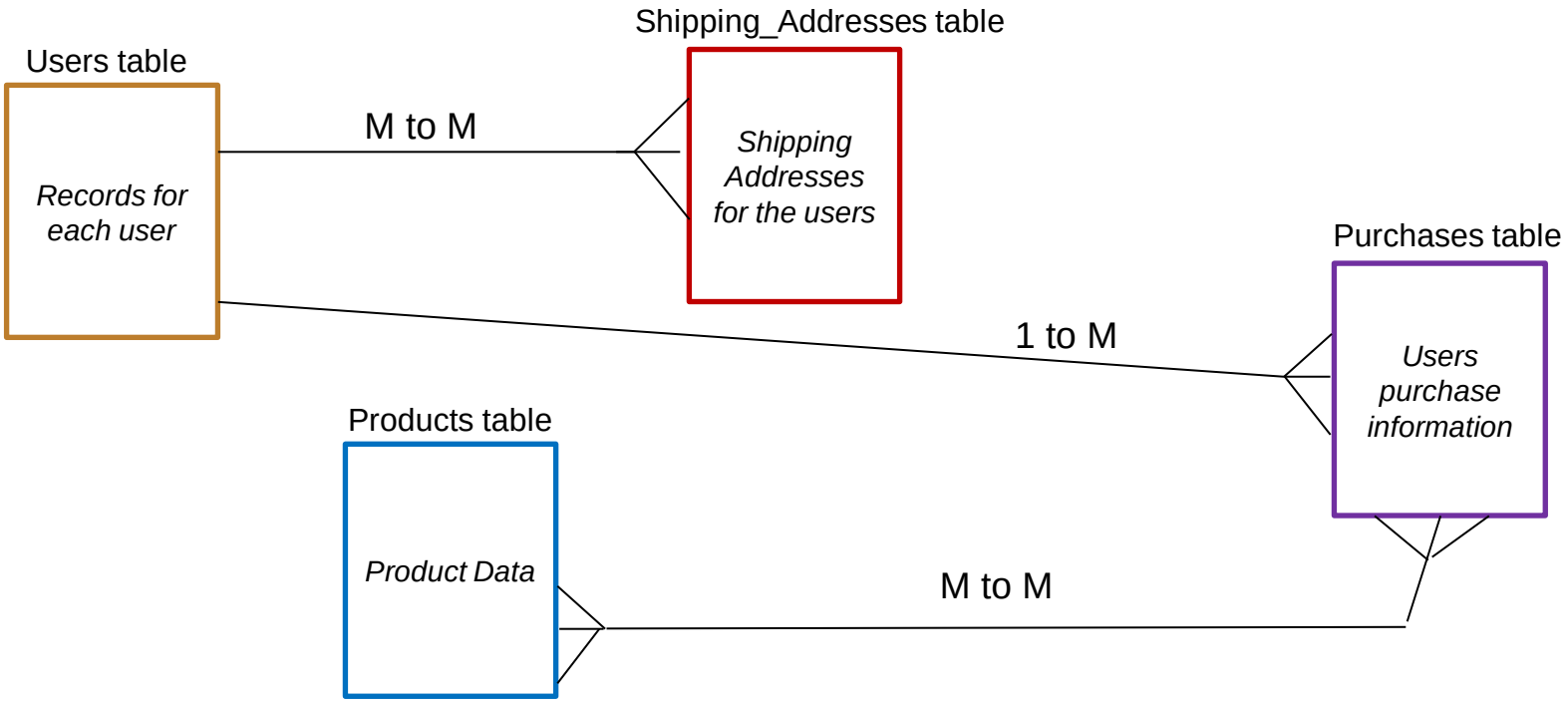
The country table's **primary key**, code, is a **foreign key** on the City table to reference what country a city is in.

# Cardinality

- Describes relationship between two tables
- Relationship between a row in one table and a row of another table.
- Options are one or many
- 1 to 1, 1 to M, M to M

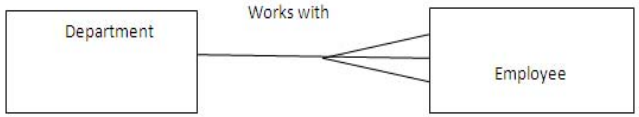


# Amazon Scenario

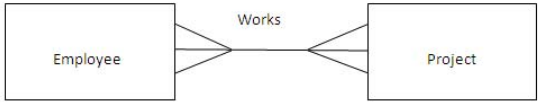


# Other examples


a one-to-one relationship




a one-to-many relationship

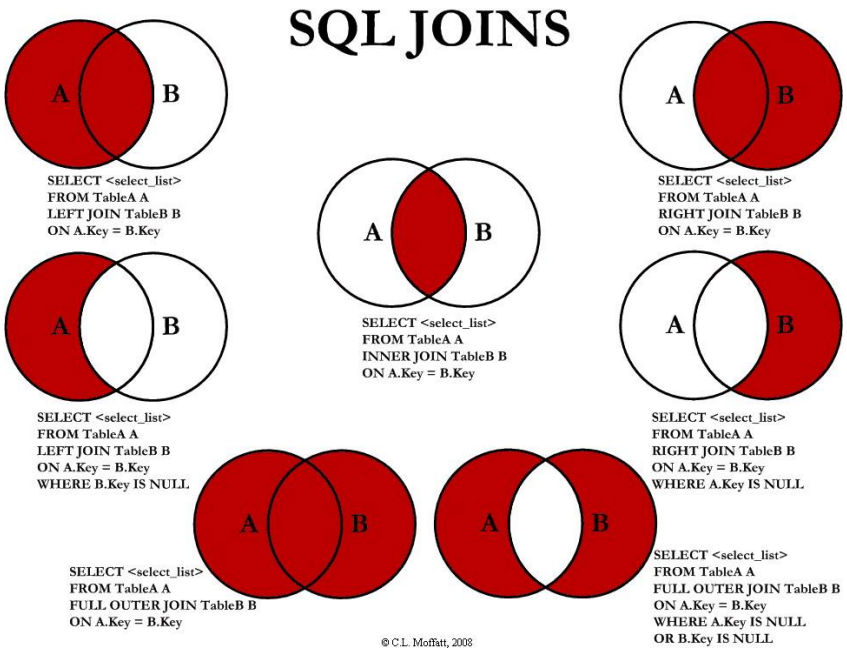



a many-to-many relationship

# Joins

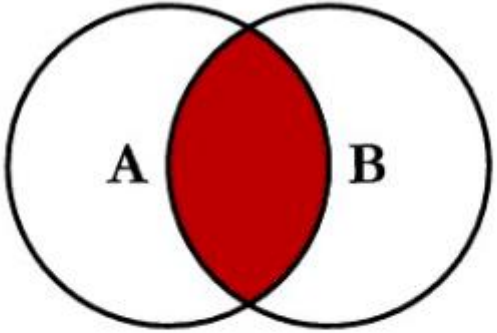
Joins in SQL allow us to pull in data from several tables.

A JOIN is an INNER JOIN



# Joins : Inner Join or Join

An inner join returns the rows in Table A that has a matching key value in Table B, the Venn Diagram representation is as follows:




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

# Joins : Inner Join Example

Consider the following example: **We need a SQL query that returns all the cities in Texas. In my result, I want to see the full state name (not the code) followed by the cities in that state**..

| | city_id [PK] integer | city_name character varying (50) | state_abbreviation character (2) | population integer | area numeric (5,1 |
|---|---|---|---|---|---|
| 1 | 1 | Abilene | TX | 123420 | 276.4 |
| 2 | 2 | Akron | OH | 197597 | 160.6 |
| 3 | 3 | Albany | NY | 96460 | 56.8 |
| 4 | 4 | Albuquerque | NM | 560513 | 487.4 |
| 5 | 5 | Alexandria | VA | 159428 | 38.8 |
| 6 | 6 | Allen | TX | 105623 | 70.2 |
| 7 | 7 | Allentown | PA | 121442 | 45.3 |
| 8 | 8 | Amarillo | TX | 199371 | 262.6 |
| 9 | 9 | Anaheim | CA | 350365 | 129.5 |
| 10 | 10 | Anchorage | AK | 288000 | 4420.1 |
| 11 | 11 | Ann Arbor | MI | 119980 | 72.8 |

- The city table contains the list of cities by state abbreviation… but it is missing the full state name.
- The state table contains the list of all states, but it has no data for cities.

Data Output Explain Messages Notifications

| | state_abbreviation [PK] character (2) | state_name character varying (50) | population integer | area integer | capital integer | sales_tax numeric (5,3) | state_nickname character varying (100) | census_region character varying (10) |
|---|---|---|---|---|---|---|---|---|
| 1 | AL | Alabama | 4903185 | 135767 | 198 | 4.000 | Heart of Dixie | South |
| 2 | AK | Alaska | 731545 | 1723337 | 155 | 0.000 | The Last Frontier | West |
| 3 | AZ | Arizona | 7278717 | 295234 | 238 | 5.600 | Grand Canyon State | West |
| 4 | AR | Arkansas | 3017804 | 137732 | 175 | 6.500 | The Natural State | South |
| 5 | CA | California | 39512223 | 423967 | 264 | 7.250 | The Golden State | West |
| 6 | CO | Colorado | 5758736 | 269601 | 88 | 2.900 | Centennial State | West |
| 7 | CT | Connecticut | 3565278 | 14357 | 133 | 6.350 | Constitution State | Northeast |
| 8 | DE | Delaware | 973764 | 6446 | 91 | 0.000 | The First State | South |

- What we need to do is combine both tables:
  - Fortunately, these two tables are “related” via the state abbreviation value. Both tables refer to the column as state_abbreviation.

# Joins : Inner Join Example

We can combine all of these facts to write a query that combines these two tables:

```
SELECT city_name AS "Texas Cities"
   FROM city c
   INNER JOIN state s
      ON c.state_abbreviation = s.state_abbreviation
   WHERE c.state_abbreviation = 'TX';
```

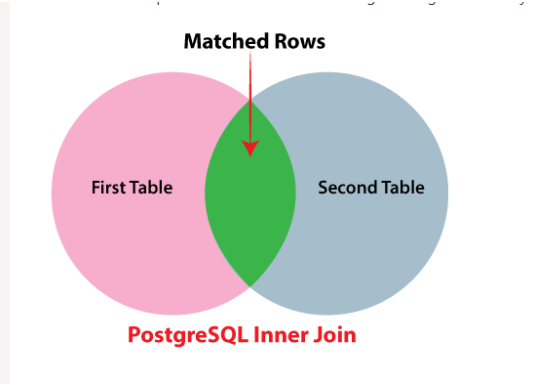
FIX!  BAD EXAMPLE!

Data Output  Explain  Messa

| | Texas Cities character varying (50) |
|---|---|
| 1 | Abilene |
| 2 | Allen |
| 3 | Amarillo |
| 4 | Arlington |
| 5 | Austin |
| 6 | Beaumont |
| 7 | Brownsville |
| 8 | Carrollton |
| 9 | College Station |
| 10 | Corpus Christi |
| 11 | Dallas |
| 12 | Denton |
| 13 | Edinburg |
| 14 | El Paso |

| luxury_cars | |
|---|---|
| l_id | INTEGER |
| luxury_car_names | CHARACTER VARYING(250) |

| sports_cars | |
|---|---|
| s_id | INTEGER |
| sports_car_names | CHARACTER VARYING(250) |




```
24
25 SELECT L_ID, luxury_car_names, S_ID, sports_car_names
26 FROM Luxury_cars
27 JOIN Sports_cars
28 ON luxury_car_names= sports_car_names;
29
30
31
```

31:1 [712] INS

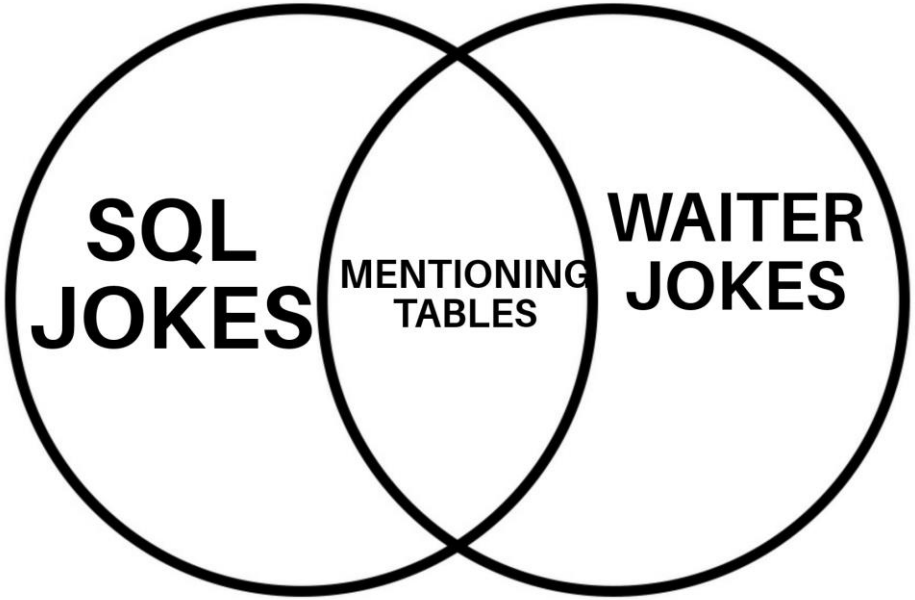
Log | 1: Luxury_cars [2]

| * | l_id | luxury_car_names | s_id | sports_car_names |
|---|---|---|---|---|
| 1 | 1 | Chevrolet Corvette | 3 | Chevrolet Corvette |
| 2 | 2 | Mercedes Benz SL Class | 4 | Mercedes Benz SL Class |

```
24
25 SELECT L_ID, luxury_car_names, S_ID, sports_car_names
26 FROM Luxury_cars
27 INNER JOIN Sports_cars
28 ON luxury_car_names= sports_car_names;
29
30
```

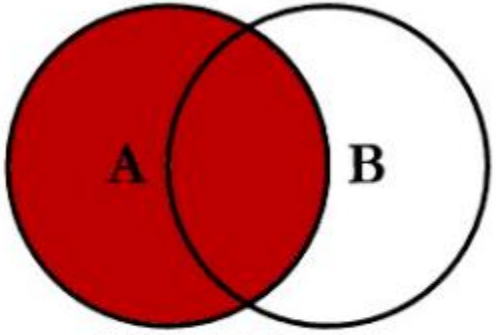
https://www.javatpoint.com/postgresql-join

# Let’s write some inner join queries!



# Joins : Left Outer Join (can also be called Left Join)

The Left Outer Join returns all the rows on the “left” side table of the join, it will attempt to match to the right side. If there is match… If it can’t find a match it includes it in the result, but with NULL values.




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

# Joins : Left Outer Join Example

```sql
SELECT state_name, COUNT(park_state.park_id) AS num_of_parks
   FROM state
   LEFT JOIN park_state
      ON state.state_abbreviation =
      park_state.state_abbreviation
   GROUP BY state_name;
```

| | state_name character varying (50) | num_of_parks bigint |
|---|---|---|
| 1 | Kansas | 0 |
| 2 | Oregon | 1 |
| 3 | Texas | 2 |
| 4 | Idaho | 1 |
| 5 | Alabama | 0 |
| 6 | Alaska | 8 |
| 7 | Oklahoma | 0 |
| 8 | North Carolina | 1 |
| 9 | Colorado | 4 |

Note that the num_of_parks for Kansas, Alabama, and Oklahoma don't have any parks in the database, so the Left Outer Join instead creates these 0 placeholders.

# Joins : Left Join vs Inner Join

With the same data set as the previous slide, let's compare the LEFT OUTER vs an INNER.

```
SELECT state_name, COUNT(park_state.park_id)
  AS num_of_parks
   FROM state
   JOIN park_state ON
      state.state_abbreviation =
      park_state.state_abbreviation
   GROUP BY state_name;
```

| | state_name character varying (50) | num_of_parks bigint |
|---|---|---|
| 1 | North Carolina | 1 |
| 2 | Colorado | 4 |
| 3 | Florida | 3 |
| 4 | Nevada | 2 |
| 5 | West Virginia | 1 |
| 6 | South Carolina | 1 |
| 7 | Arkansas | 1 |
| 8 | Hawaii | 2 |
| 9 | New Mexico | 2 |

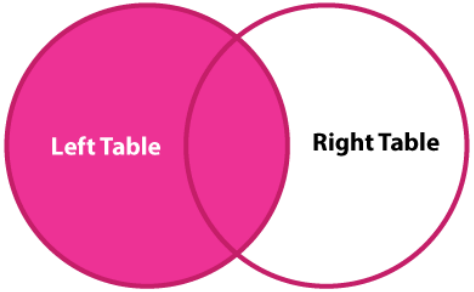
With the INNER JOIN, the rows for which there are no matches on the key are dropped from the final result set.

```
SELECT state_name, COUNT(park_state.park_id) AS num_of_parks
   FROM state
   LEFT JOIN park_state
      ON state.state_abbreviation =
      park_state.state_abbreviation
   GROUP BY state_name;
```

| | state_name character varying (50) | num_of_parks bigint |
|---|---|---|
| 1 | Kansas | 0 |
| 2 | Oregon | 1 |
| 3 | Texas | 2 |
| 4 | Idaho | 1 |
| 5 | Alabama | 0 |
| 6 | Alaska | 8 |
| 7 | Oklahoma | 0 |
| 8 | North Carolina | 1 |
| 9 | Colorado | 4 |

| luxury_cars | |
|---|---|
| l_id | INTEGER |
| luxury_car_names | CHARACTER VARYING(250) |

| sports_cars | |
|---|---|
| s_id | INTEGER |
| sports_car_names | CHARACTER VARYING(250) |




PostgreSQL Left Join

```
30
31
32 SELECT L_ID, luxury_car_names, S_ID, sports_car_names
33 FROM Luxury_cars
34 LEFT JOIN Sports_cars
35 ON luxury_car_names= sports_car_names;
36
37
```

35:36 [853] INS

Log | 1: Luxury_cars [5]

| * | l_id | luxury_car_names | s_id | sports_car_names |
|---|---|---|---|---|
| 1 | 1 | Chevrolet Corvette | 3 | Chevrolet Corvette |
| 2 | 2 | Mercedes Benz SL Class | 4 | Mercedes Benz SL Class |
| 3 | 3 | Audi A7 | (null) | (null) |
| 4 | 4 | Genesis G90 | (null) | (null) |
| 5 | 5 | Lincoln Continental | (null) | (null) |

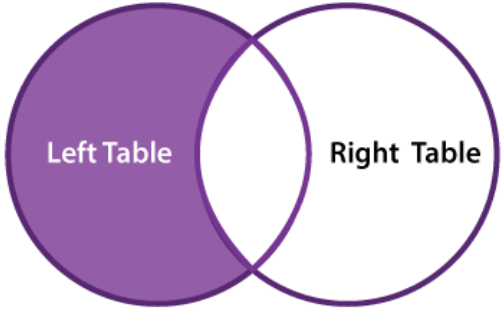
```
31
32 SELECT L_ID, luxury_car_names, S_ID, sports_car_names
33 FROM Luxury_cars
34 LEFT OUTER JOIN Sports_cars
35 ON luxury_car_names= sports_car_names;
36
37
```

https://www.javatpoint.com/postgresql-join

| luxury_cars | |
|---|---|
| l_id | INTEGER |
| luxury_car_names | CHARACTER VARYING(250) |

| sports_cars | |
|---|---|
| s_id | INTEGER |
| sports_car_names | CHARACTER VARYING(250) |

**Left Outer Join: Select Rows From the Left table**




```
45
46 SELECT L_ID, luxury_car_names, S_ID, sports_car_names
47 FROM Luxury_cars
48 LEFT JOIN Sports_cars
49 ON luxury_car_names= sports_car_names
50 WHERE S_ID IS NULL;
```

50:22 [1191] INS

Log | 1: Luxury_cars [3]

| * | l_id | luxury_car_names | s_id | sports_car_names |
|---|---|---|---|---|
| 1 | 3 | Audi A7 | (null) | (null) |
| 2 | 4 | Genesis G90 | (null) | (null) |
| 3 | 5 | Lincoln Continental | (null) | (null) |

https://www.javatpoint.com/postgresql-join

# Let’s write some left join queries!

# Popular Interview Question...

Question: What is the difference between a **LEFT JOIN** and a **LEFT OUTER JOIN**?

Answer: *Nothing, they are the same!*

Question: What is the difference between a **RIGHT JOIN** and **RIGHT OUTER JOIN**?

Answer: *Nothing, they are the same!*

# Unions

A union is a combination of two result sets. The following pattern is used:

**[SQL Query 1]**

**UNION**

**[SQL Query 2]**

# Unions Example

Consider the following query:

```sql
SELECT city_name AS place_name,
   'City' AS kind_of_place
    FROM city WHERE city_name LIKE 'W%'
UNION
SELECT state_name AS place_name,
   'State' AS kind_of_place
   FROM state WHERE state_name LIKE 'W%'
ORDER BY place_name;
```

| | place_name character varying (50) | kind_of_place text |
|---|---|---|
| 1 | Waco | City |
| 2 | Warren | City |
| 3 | Washington | City |
| 4 | Washington | State |
| 5 | Waterbury | City |
| 6 | West Covina | City |
| 7 | West Jordan | City |
| 8 | West Palm Beach | City |
| 9 | West Valley City | City |
| 10 | West Virginia | State |
| 11 | Westminster | City |

Result of query first query (returns the City)

Result of query second query (returns the State)

# Union All

## 2) PostgreSQL UNION ALL example

The following statement uses the `UNION ALL` operator to combine result sets from the `top_rated_films` and `most_popular_films` tables:

```
SELECT * FROM top_rated_films
UNION ALL
SELECT * FROM most_popular_films;
```

| | title<br>character varying | release_year<br>smallint |
|---|---|---|
| 1 | The Shawshank Redemption | 1994 |
| 2 | The Godfather | 1972 |
| 3 | 12 Angry Men | 1957 |
| 4 | An American Pickle | 2020 |
| 5 | The Godfather | 1972 |
| 6 | Greyhound | 2020 |

In this example, the duplicate row is retained in the result set.

| most_reliable_cars | |
|---|---|
| car_name | CHARACTER VARYING |
| launch_year | SMALLINT |

| top_rated_cars | |
|---|---|
| car_name | CHARACTER VARYING |
| launch_year | SMALLINT |

```
SELECT * FROM top_rated_cars;
```

82:30 [1763] INS

Log | 1: top_rated_cars [3]

| * | car_name | launch_year |
|---|---|---|
| 1 | Chevrolet Silveradon | 2020 |
| 2 | Nissan Rogue | 2020 |
| 3 | Mercedes-Benz GLB | 2019 |

```
SELECT * FROM most_reliable_cars;
```

84:34 [1798] INS

Log | 1: most_reliable_cars [3]

| * | car_name | launch_year |
|---|---|---|
| 1 | Toyota Prius Prime | 2020 |
| 2 | Nissan Rogue | 2020 |
| 3 | Kia Forte | 2019 |

```
SELECT * FROM top_rated_cars
UNION
SELECT * FROM most_reliable_cars;
```

89:36 [1875] INS

Log | 1: top_rated_cars [5]

| * | car_name | launch_year |
|---|---|---|
| 1 | Chevrolet Silveradon | 2020 |
| 2 | Mercedes-Benz GLB | 2019 |
| 3 | Nissan Rogue | 2020 |
| 4 | Toyota Prius Prime | 2020 |
| 5 | Kia Forte | 2019 |

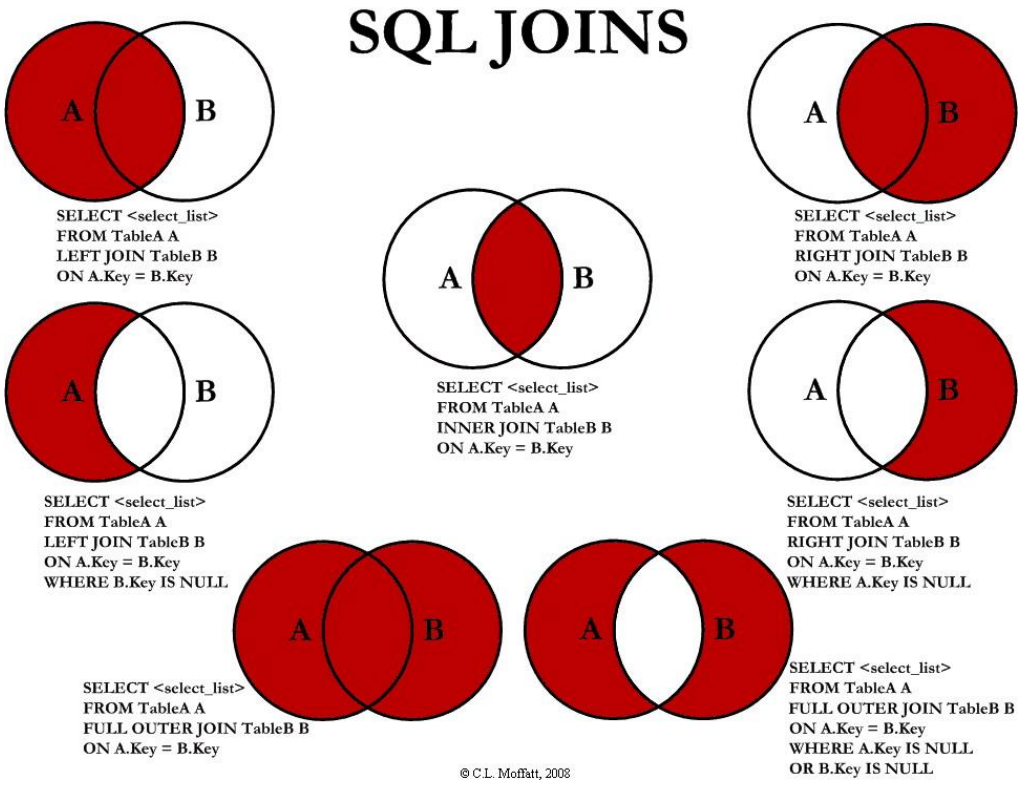
```
SELECT * FROM top_rated_cars
UNION ALL
SELECT * FROM most_reliable_cars;
```

94:36 [1956] INS

Log | 1: top_rated_cars [6]

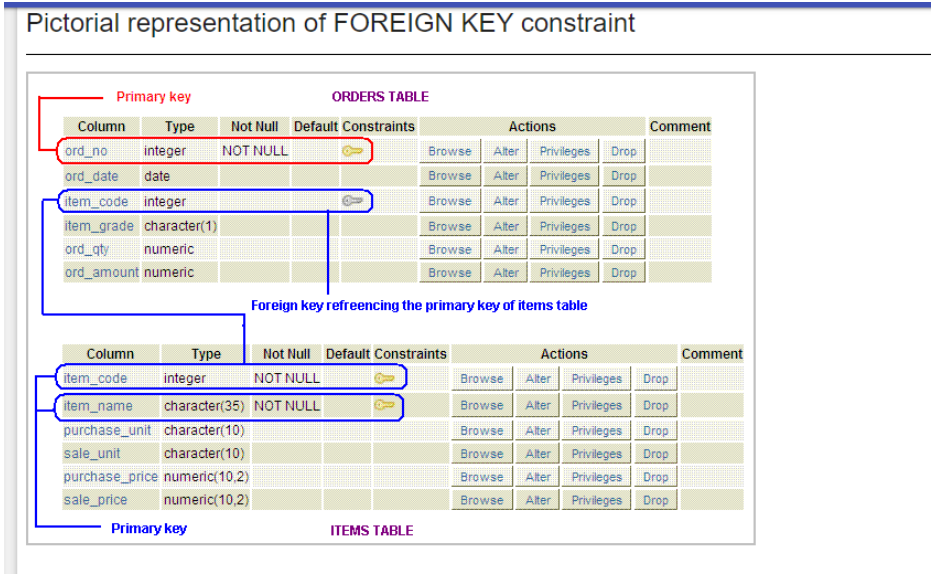
| * | car_name | launch_year |
|---|---|---|
| 1 | Chevrolet Silveradon | 2020 |
| 2 | Nissan Rogue | 2020 |
| 3 | Mercedes-Benz GLB | 2019 |
| 4 | Toyota Prius Prime | 2020 |
| 5 | Nissan Rogue | 2020 |
| 6 | Kia Forte | 2019 |

# Joins


SQL JOINS
A
B
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
© C.L. Moffatt, 2008

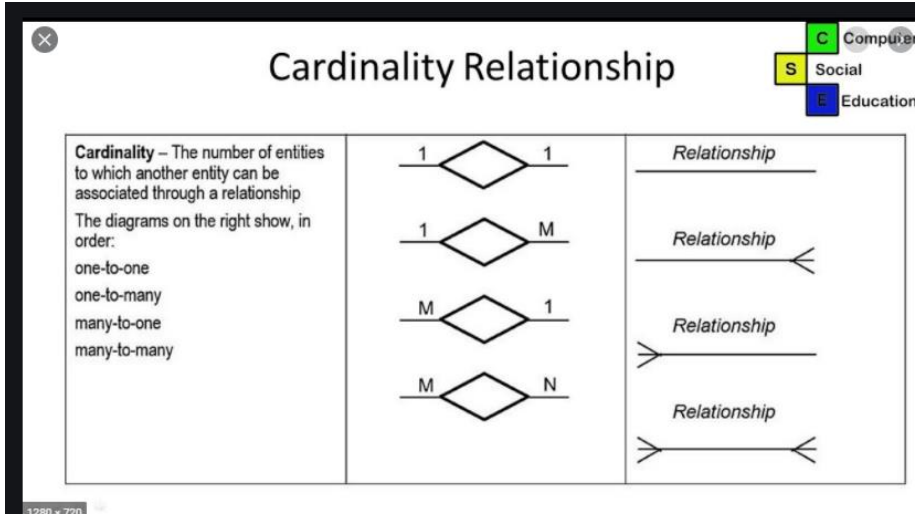
# Objectives

- Keys (Primary, Natural, Surrogate, Foreign)



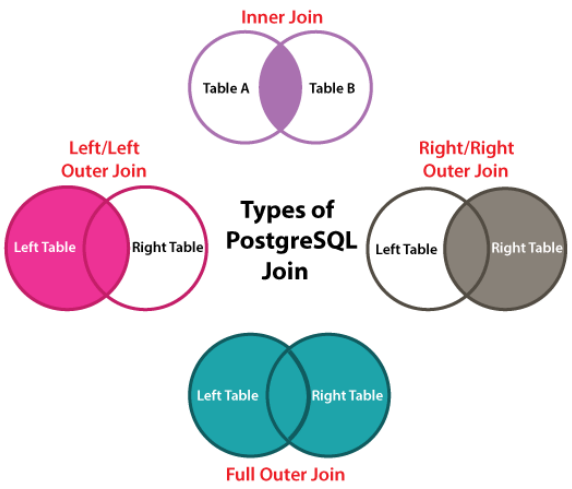
# Objectives

- Keys (Primary, Natural, Surrogate, Foreign)
- Cardinality (1-1, 1-M, M-M)



# Objectives

- Keys (Primary, Natural, Surrogate, Foreign)
- Cardinality (1-1, 1-M, M-M)
- SQL Joins



# Objectives

- Keys (Primary, Natural, Surrogate
- Cardinality (1-1, 1-M, M-M)
- SQL Joins
- Unions

We have a database named Demo with the following tables:

**Book:**

```
Demo=# select * from book;
 id ¦                name
----+-----------------------------------
  1 ¦ PostgreSQL for Beginners
  2 ¦ PostgreSQL, A Step-by-Step Guide
  3 ¦ PostgreSQL for Advanced Users
  4 ¦ Learn PostgreSQL in 7 Days
  5 ¦ PostgreSQL Made Easy
(5 rows)

Demo=#
```

**Price:**

```
Demo=# SELECT * FROM Price;
 id ¦ price
----+-------
  1 ¦   200
  2 ¦   250
  3 ¦   220
  4 ¦   190
  5 ¦   300
(5 rows)
```

Let us run the following command:

```
SELECT id
FROM Book
UNION
SELECT id
FROM Price;
```

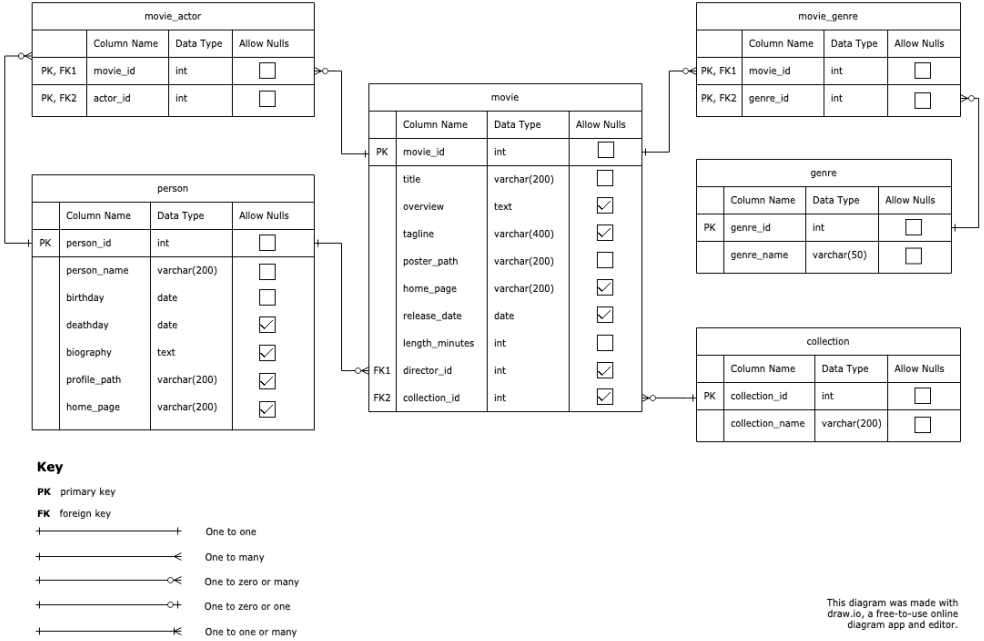
The command will return the following:

```
Demo=# SELECT id
Demo-# FROM Book
Demo-# UNION
Demo-# SELECT id
Demo-# FROM Price;
 id
----
  2
  5
  4
  3
  1
(5 rows)

Demo=#
```

# Objectives

- Keys (Primary, Natural, Surrogate, Foreign)
- Cardinality (1-1, 1-M, M-M)
- SQL Joins
- Unions
- Create a new Database



# Let’s write some unions!