## ***SELECT FROM***

- Query data from all columns, can use an asterisk (*) as the shorthand for all columns
- SQL language is capital insensitive

Syntax:
**SELECT** column1, colunm2, … **FROM** table_name

**SELECT** * **FROM** actor
# Select all the columns from actor

**SELECT** first_name, last_name **FROM** actor
Select specific column (first_name, last_name) from actor

## ***SELECT DISTINCT***

- In a table, a column may contain many duplicate values; and sometimes you only want to list the different (distinct) values.
- The DISTINCT keyword can be used to return only distinct (different) values.

Syntax:
**SELECT DISTINCT** column_1, column_2 **FROM** table_name

## ***SELECT WHERE***

Query just particular rows from a table

Syntax:
**SELECT** column_1, column_2 … column_n **FROM** table_name
**WHERE** conditions

Operators:
Equal (=), greater than (>), less than (<), greater than or equal (>=), less than or equal (<=), not equal (<> /! =), AND, OR

- If you want to get all customers whose first names are Jamie, you can use the WHERE with the **equal (=)** operator as follows:

**SELECT** last_name, first_name **FROM** customer
**WHERE** first_name = 'Jamie'

- If you want to select the customer whose first name is Jamie and the last names is Rice, you can use the **AND** logical operator that combines two conditions as the following query:

**SELECT** last_name, first_name **FROM** customer
**WHERE** first_name = 'Jamie' **AND** last_name = 'Rice'

- If you want to know who paid the rental with amount is either less than $1 or greater than $8, you can use the following query with OR operator:

**SELECT** customer_id, amount, payment_date **FROM** payment
**WHERE** amount <= 1 **OR** amount >= 8

## ***COUNT***

- The COUNT (*) function returns the number of rows returned by a SELECT clause.
- When you apply the COUNT (*) to the entire table, PostgreSQL scans table sequentially.

Syntax:
**SELECT COUNT** (*) **FROM** table_column (it doesn't consider NULL values in the column)

**SELECT COUNT** (*) **FROM** payment
# count number of rows of payment

**SELECT COUNT** (**DISTINCT** amount) **FROM** payment
# number of distinct types in the amount row

## ***LIMIT***

- LIMIT allows you to limit the number of rows you get back after a query.
- Useful when wanting to get all columns but not all rows.
- Goes at the end of a query

Syntax:
**SELECT** * **FROM** customer
**LIMIT** 5
# return first 5 rows of customer table

## ***ORDER BY***

- When you query data from a table, PostgreSQL returns the rows in the order that they were inserted into the table
- In order to sort the result set, you use the ORDER BY clause in the SELECT statement in ascending or descending order based on criteria specified.
- Specify the column that you want to sort in the ORDER BY clause. If you sort the result set by multiple columns, use a comma to separate between two columns.
- Use ASC to sort the result set in ascending order and DESC to sort the result set descending order.
- If you leave it blank, the ORDER BY clause will use ASC by default.

Syntax:
**SELECT** column_1, column_2 **FROM** table_name
**ORDER BY** column_1 **ASC / DESC**

**SELECT** first_name, last_name **FROM** customer
**ORDER BY** first_name **ASC**
# returned first name and last name and order from A to Z in first name

**SELECT** first_name, last_name **FROM** customer
**ORDER BY** first_name **ASC**, last_name **DESC**
# first names are ordered from A to Z, if there are same names, then their last names are ordered from Z to A

**SELECT** first_name, last_name **FROM** customer
**ORDER BY** last_name
Equal to:
**SELECT** first_name **FROM** customer
**ORDER BY** last_name
PostgreSQL has the ability to sort result sets based on columns that do not appear in the selection list.

## ***WHERE BETWEEN/NOT BETWEEN***

- If the value is greater than or equal to the low value and less than or equal to the high value, the expression returns true, or vice versa.
- We can rewrite the BETWEEN operator by using the greater than or equal (>=) or less than or equal (<=) operators as the following statement:
**Value** >= low **and value** <= high;
Using BETWEEN operator:
**Value BETWEEN** low **AND** high;

**SELECT** customer_id, amount **FROM** payment
**WHERE** amount **BETWEEN** 8 **AND** 9
# Returns amount columns which the amount is between 8 and 9

## ***WHERE IN/NOT IN***

- The expression returns true if the value matches any value in the list i.e., value1, value2, etc.
- The list of values is not limited to a list of numbers or strings but also a result set of a SELECT statement as shown in the following query:

**Value IN** (**SELECT** value **FROM** table_name)

**SELECT** customer_id, rental_id, return_date **FROM** rental
**WHERE** customer_id **IN** (1,10,13)
**ORDER BY** return_date **ASC**
# Returns the customer's ID is 1, 10, and 13 and ordered from start date to end date

## ***LIKE/NOT LIKE*** **(LIKE is capital sensitivity, ILIKE is capital insensitivity)**

Suppose the store manager asks you find a customer that he does not remember the name exactly. He just remembers that customer's first name begins with something like Jen. How do you find the exact customer that the store manager is asking?

**SELECT** first_name, last_name **FROM** customer
**WHERE** first_name **LIKE** 'Jen%'

Notice that the WHERE clause contains a special expression:

The first_name, the LIKE operator and a string that contains a percent (%) character, which is referred as a pattern.

The query returns rows whose values in the first name column begin with Jen and may be followed by any sequence of characters. This technique is called pattern matching.

- Percent (%) for matching any sequence of characters
- Underscore (_) for matching any single character

**SELECT** first_name, last_name **FROM** customer
**WHERE** first_name **LIKE** '_er%'
# Returns any first name with 'single character'+ 'er' + 'any sequence of characters'
Such as: Teresa, Bertha, Veronica…

## ***Questions***

1. How many payment transactions were greater than $5?

```
SELECT COUNT (amount) FROM payment
WHERE amount > 5
```

2. How many actors have a first name that start with the letter P?

```
SELECT COUNT (first_name) FROM actor
WHERE first_name LIKE 'P%'
```

3. How many unique districts are our customers from?

```
SELECT COUNT (DISTINCT district) FROM address
```

4. Retrieve the list of names for those distinct districts from the previous question

```
SELECT DISTINCT district FROM address
```

5. How many films have a rating of R and a replacement cost between $5 and $15?

```
SELECT * FROM film
WHERE rating = 'R'
AND replacement_cost BETWEEN 5 AND 15
```

6. How many films have the word Truman somewhere in the title?

```
SELECT COUNT (*) FROM film
WHERE title LIKE '%Truman%'
```

## ***MIN/MAX/AVG/SUM***

**SELECT AVG** (amount) **FROM** payment
# 4.2006056453822965
**SELECT ROUND** (**AVG** (amount), 2) **FROM** payment
# 4.20
**SELECT ROUND** (**AVG** (amount), 5) **FROM** payment
# 4.20061
**SELECT MIN** (amount) **FROM** payment
# 0.00
**SELECT MAX** (amount) **FROM** payment
# 11.99
**SELECT SUM** (amount) **FROM** payment
# 61312.04

## ***GROUP BY*** ***(when use GROUP BY, be aware of using aggregate function)***

- The GROUP BY clause divides the rows returned from the SELECT statement into groups.
- For each group, you can apply an aggregate function, for example;
    - calculating the sum of items
    - count the number of items in the groups

Syntax:
**SELECT** column_1, aggregate_function (column_2) **FROM** table_name
**GROUP BY** column_1

**SELECT** customer_id **FROM** payment
**GROUP BY** customer_id
# (same as DISTINCT) returns 599 distinct customer id, customer's ids are grouped together

**SELECT** customer_id, **SUM** (amount) **FROM** payment
**GROUP BY** customer_id
# for every customer ID, group the customer ID and then sum that specific customer's amount column. Ex: for customer-id is 251, the total amount is 100.75

**SELECT** rating, **COUNT** (rating) **FROM** film
**GROUP BY** rating
# aggregating the counts of the different rating types

## ***Questions***

1. We have two staff members with Staff IDs 1 and 2. We want to give a bonus to the staff member that handle the most payments. How many payments did each staff member handle? And how much was the total amount processed by each staff member?

```
SELECT staff_id, SUM (amount), COUNT (amount) FROM payment
GROUP BY staff_id
```

2. Corporate headquarters is auditing our store! They want to know the average replacement cost of movies by rating. For example, R rated movies have an average replacement cost of $20.23.

```
SELECT rating, ROUND (AVG (replacement_cost),2) FROM film
GROUP BY rating
```

3. We want to send coupons to the 5 customers who have spent the most amount of money. Give me the customer ids of the top 5 spenders.

```
SELECT customer_id, SUM (amount) FROM payment
GROUP BY customer_id
ORDER BY SUM (amount) DESC
LIMIT 5
```

## ***HAVING with GROUP BY***

- We often use the HAVING clause in conjunction with the GROUP BY clause to filter group rows that do not satisfy a specified condition.
- The HAVING clause sets the condition for group rows created by the GROUP BY clause after the GROUP BY clause applies while the WHERE clause sets the condition for individual rows before GROUP BY clause applies. This is the main difference between the HAVING and WHERE clause.
- HAVING is used after GROUP BY, WHERE is used before GROUP BY

Syntax

**SELECT** column_1, aggregate_function (column_2) **FROM** table_name
**GROUP BY** column_1 **HAVING** condition

**SELECT** customer_id, **SUM** (amount) **FROM** payment
**GROUP BY** customer_id
**HAVING SUM** (amount) > 200
# returns customer ids whose amount are larger than $200

**SELECT** rating, **AVG** (rental_rate) **FROM** film
**WHERE** rating IN ('R','G','PG')
**GROUP BY** rating
# returns rows of R, G, and PG and their average rates

## ***QUESTIONS***

1. We want to know what customers are eligible for our platinum credit. The requirements are that the customer has at least a total of 40 transaction payments. What customers (by customer_id) are eligible for the credit card?

```
SELECT customer_id, COUNT (amount) FROM payment
GROUP BY customer_id
HAVING COUNT (amount) >= 40
```

2. When grouped by rating, what movie ratings have an average rental duration of more than 5 days?

```
SELECT rating, AVG (rental_duration) FROM film
GROUP BY rating
HAVING AVG (rental_duration) > 5
```

## ***ASSESSMENT TEST 1***

1. Return the customer IDs of customers who have spent at least $110 with the staff member who has an ID of 2.

```
SELECT customer_id, SUM (amount) FROM payment
WHERE staff_id = 2
GROUP BY customer_id
HAVING SUM (amount) > 110
```

2. How many films begin with the letter J?

```
SELECT COUNT (title) FROM film
WHERE title LIKE 'J%'
```

3. What customer has the highest customer ID number whose name starts **with** an 'E' **and** has an address ID lower than 500?

```
SELECT first_name, last_name FROM customer
WHERE first_name LIKE 'E%'
AND address_id < 500
ORDER BY customer_id DESC
LIMIT 1
```

## ***Joins***

Assume we have the following two tables. **Table A** is on the left, and **Table B** is on the right. We'll populate them with four records each.

```
id name       id  name
-- ----       --  ----
1  Pirate     1   Rutabaga
2  Monkey     2   Pirate
3  Ninja      3   Darth Vader
4  Spaghetti  4   Ninja
```

Let's join these tables by the name field in a few different ways and see if we can get a conceptual match to those nifty Venn diagrams.

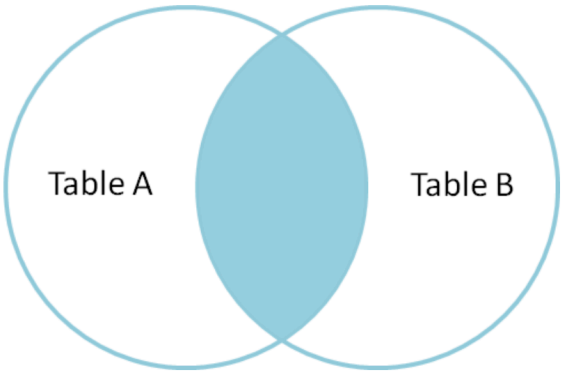
**Inner join** produces only the set of records that match in both Table A and Table B.

```
SELECT * FROM TableA
INNER JOIN TableB
ON TableA.name = TableB.name

id  name       id   name


1   Pirate     2    Pirate

3   Ninja      4    Ninja
```

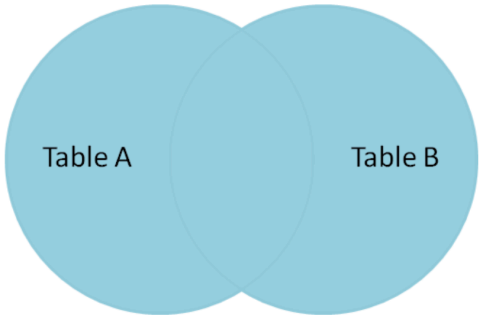


```
id name       id  name
-- ----       --  ----
1  Pirate     1   Rutabaga
2  Monkey     2   Pirate
3  Ninja      3   Darth Vader
4  Spaghetti  4   Ninja
```

**Full outer join** produces the set of all records in Table A and Table B, with matching records from both sides where available. If there is no match, the missing side will contain null.

```
SELECT * FROM TableA
FULL OUTER JOIN TableB
ON TableA.name = TableB.name

id    name       id    name
---------------------------
1     Pirate     2     Pirate
2     Monkey     null  null
3     Ninja      4     Ninja
4     Spaghetti  null  null
null  null       1     Rutabaga
null  null       3     Darth Vader
```

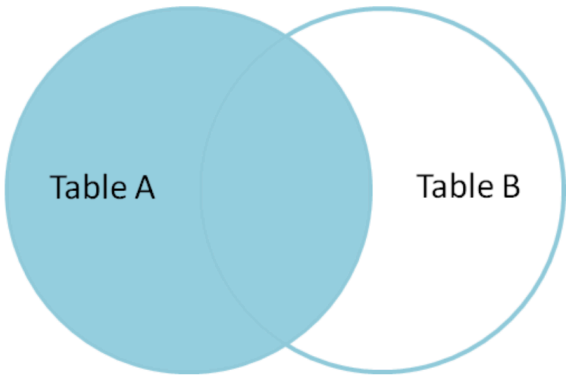


```
id name       id  name
-- ----       --  ----
1  Pirate     1   Rutabaga
2  Monkey     2   Pirate
3  Ninja      3   Darth Vader
4  Spaghetti  4   Ninja
```

**Left outer join** produces a complete set of records from Table A, with the matching records (where available) in Table B. If there is no match, the right side will contain null.

```
SELECT * FROM TableA
LEFT OUTER JOIN TableB
ON TableA.name = TableB.name

id  name       id    name
--  ----       --    ----
1   Pirate     2     Pirate
2   Monkey     null  null
3   Ninja      4     Ninja
4   Spaghetti  null  null
```



```
id name       id  name
-- ----       --  ----
1  Pirate     1   Rutabaga
2  Monkey     2   Pirate
3  Ninja      3   Darth Vader
4  Spaghetti  4   Ninja
```

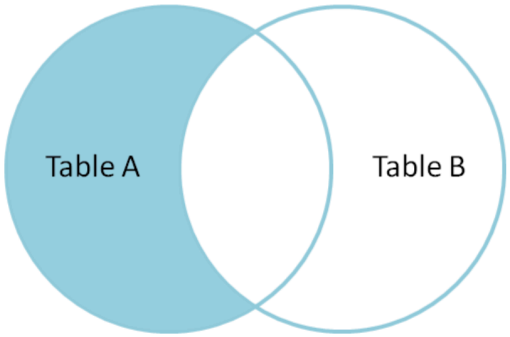
To produce the set of records only in Table A, but not in Table B, we perform the same left outer join, then **exclude the records we don't want from the right side via a where clause**.

```
SELECT * FROM TableA
LEFT OUTER JOIN TableB
ON TableA.name = TableB.name
WHERE TableB.id IS null

id  name       id    name
```

```
2   Monkey     null  null

4   Spaghetti  null  null
```



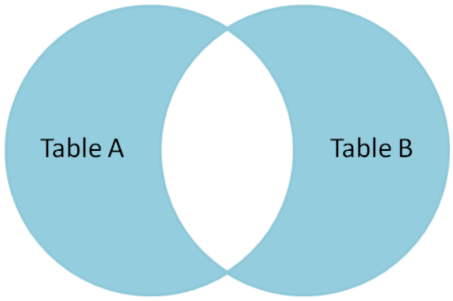
```
id name       id  name
-- ----       --  ----
1  Pirate     1   Rutabaga
2  Monkey     2   Pirate
3  Ninja      3   Darth Vader
4  Spaghetti  4   Ninja
```

To produce the set of records unique to Table A and Table B, we perform the same full outer join, then **exclude the records we don't want from both sides via a where clause**.

```
SELECT * FROM TableA
FULL OUTER JOIN TableB
ON TableA.name = TableB.name
WHERE TableA.id IS null
OR TableB.id IS null

id    name       id    name
```

```
2     Monkey     null  null
4     Spaghetti  null  null
null  null       1     Rutabaga
null  null       3     Darth Vader
```




There's also a cartesian product or **cross join**, which as far as I can tell, can't be expressed as a Venn diagram:

```
SELECT * FROM TableA
CROSS JOIN TableB
```

## ***AS***

**AS** allows us to rename column or table selections with an alias

**SELECT** customer_id, **SUM** (amount) **AS** total_spent
**FROM** payment
**GROUP BY** customer_id

| | customer_id smallint | total_spent numeric |
|---|---|---|
| 1 | 184 | 80.80 |
| 2 | 87 | 137.72 |
| 3 | 477 | 106.79 |
| 4 | 273 | 130.72 |

## ***Joins***

There are several kinds of joins, including INNER JOIN, OUTER JOIN (LEFT, RIGHT), FULL JOIN, and SELF-JOIN.

**INNER JOIN**

- first, you specify the column in both tables from which you want to select data in the SELECT clause
- second, you specify the main table i.e., TableA in the FROM clause
- third, you specify the table that the main table joins to i.e., TableB in the INNER JOIN clause. In addition, you put a join condition after the ON keyword i.e., TableA.colname = TableB.colname.

**SELECT** * **FROM** TableA
**INNER JOIN** TableB
**ON** TableA.colname = TableB.colname

- For each row in the A table, PostgreSQL scans the B table to check if there is any row that matches the condition i.e., TableA.colname = TableB.colname
- If it finds a match, it combines columns of both row into one row and add the combined row to the returned result set.
- Sometimes A and B tables have the same column name so we have to refer to the column as table_name.column_name to avoid ambiguity.
- In case the name of the table is long, you can use a table alias e.g., tbland refer to the column as tbl.column_name

**SELECT** customer.customer_id, first_name, last_name, email, amount, payment_date
**FROM** customer
**INNER JOIN** payment **ON** payment.customer_id = customer.customer_id

| | customer_id integer | first_name character varying (45) | last_name character varying (45) | email character varying (50) | amount numeric (5,2) | payment_date timestamp without time zone |
|---|---|---|---|---|---|---|
| 1 | 341 | Peter | Menard | peter.menard@sakilacustom… | 7.99 | 2007-02-15 22:25:46.996577 |
| 2 | 341 | Peter | Menard | peter.menard@sakilacustom… | 1.99 | 2007-02-16 17:23:14.996577 |
| 3 | 341 | Peter | Menard | peter.menard@sakilacustom… | 7.99 | 2007-02-16 22:41:45.996577 |

**SELECT** customer.customer_id, first_name, last_name, email, amount, payment_date
**FROM** customer
**INNER JOIN** payment **ON** payment.customer_id = customer.customer_id
**ORDER BY** customer.customer_id *(there is customer_id in Payment table, so it should be point out it's customer_id in Customer table)*

| | customer_id integer | first_name character varying (45) | last_name character varying (45) | email character varying (50) | amount numeric (5,2) | payment_date timestamp without time zone |
|---|---|---|---|---|---|---|
| 1 | 1 | Mary | Smith | mary.smith@sakilacustomer… | 5.99 | 2007-02-14 23:22:38.996577 |
| 2 | 1 | Mary | Smith | mary.smith@sakilacustomer… | 0.99 | 2007-02-15 16:31:19.996577 |
| 3 | 1 | Mary | Smith | mary.smith@sakilacustomer… | 9.99 | 2007-02-15 19:37:12.996577 |

**SELECT** customer.customer_id, first_name, last_name, email, amount, payment_date
**FROM** customer
**INNER JOIN** payment **ON** payment.customer_id = customer.customer_id
**WHERE** customer.customer_id = 2

| | customer_id integer | first_name character varying (45) | last_name character varying (45) | email character varying (50) | amount numeric (5,2) | payment_date timestamp without time zone |
|---|---|---|---|---|---|---|
| 1 | 2 | Patricia | Johnson | patricia.johnson@sakilacust… | 2.99 | 2007-02-17 19:23:24.996577 |
| 2 | 2 | Patricia | Johnson | patricia.johnson@sakilacust… | 0.99 | 2007-03-01 08:13:52.996577 |
| 3 | 2 | Patricia | Johnson | patricia.johnson@sakilacust… | 0.99 | 2007-03-02 00:39:22.996577 |

**SELECT** customer.customer_id, first_name, last_name, email, amount, payment_date
**FROM** customer
**INNER JOIN** payment **ON** payment.customer_id = customer.customer_id
**WHERE** first_name **LIKE** 'A%'

| | customer_id integer | first_name character varying (45) | last_name character varying (45) | email character varying (50) | amount numeric (5,2) | payment_date timestamp without time zone |
|---|---|---|---|---|---|---|
| 1 | 346 | Arthur | Simpkins | arthur.simpkins@sakilacust… | 5.99 | 2007-02-17 09:35:32.996577 |
| 2 | 346 | Arthur | Simpkins | arthur.simpkins@sakilacust… | 2.99 | 2007-02-21 12:02:45.996577 |
| 3 | 346 | Arthur | Simpkins | arthur.simpkins@sakilacust… | 2.99 | 2007-02-21 15:51:24.996577 |

```
SELECT payment_id, amount, first_name, last_name
FROM payment
INNER JOIN staff ON payment.staff_id = staff.staff_id
```

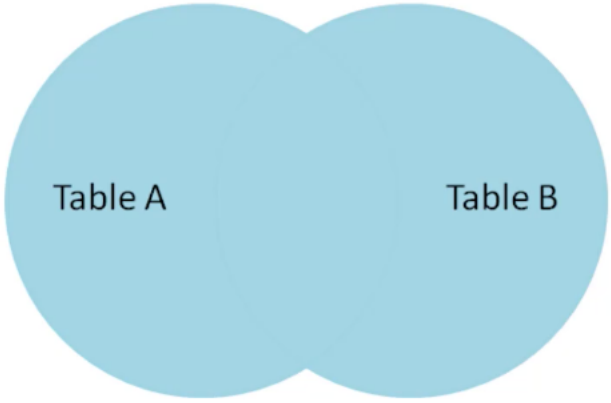
| | payment_id integer | amount numeric (5,2) | first_name character varying (45) | last_name character varying (45) |
|---|---|---|---|---|
| 1 | 17503 | 7.99 | Jon | Stephens |
| 2 | 17504 | 1.99 | Mike | Hillyer |
| 3 | 17505 | 7.99 | Mike | Hillyer |

```
SELECT title, COUNT (title) AS copes_at_store1 FROM inventory
INNER JOIN film ON inventory.film_id = film.film_id
WHERE store_id = 1
GROUP BY title
ORDER BY title
```

| | title character varying (255) | copes_at_store1 bigint |
|---|---|---|
| 1 | Academy Dinosaur | 4 |
| 2 | Affair Prejudice | 4 |
| 3 | Agent Truman | 3 |

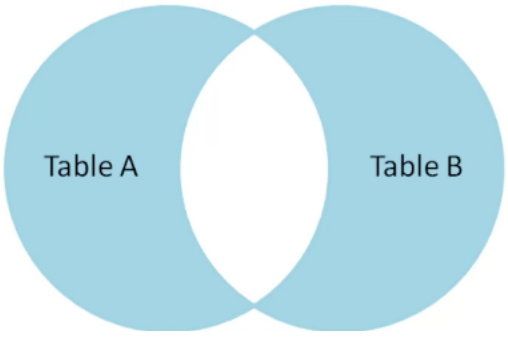
**FULL OUTER JOIN**

**SELECT** * **FROM** TableA
**FULL OUTER JOIN** TableB
**ON** TableA.colmatch = TableB.colmatch



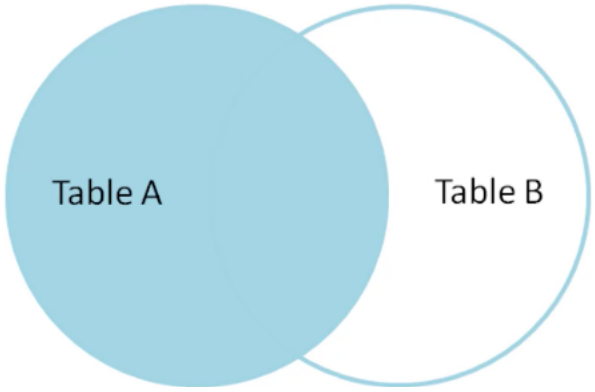

**FULL OUTER JOIN** with **WHERE**

**SELECT** * **FROM** TableA
**FULL OUTER JOIN** TableB
**ON** TableA.colmatch = TableB.colmatch
**WHERE** TableA.col1 IS NULL
**OR** TableB.col2 IS NULL



**LEFT OUTER JOIN** results in the set of records that are in the left table, if there is no match with the right table, the results are null. (Order matters for Left Outer Join)

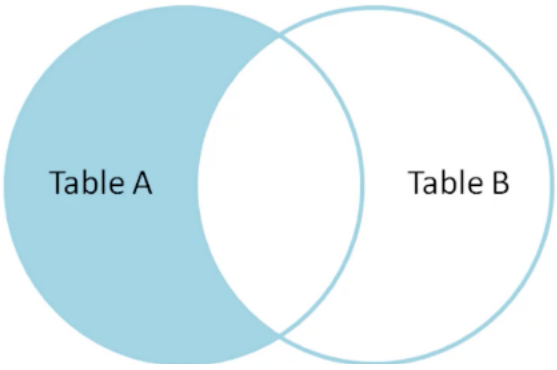
**SELECT** * **FROM** TableA
**LEFT OUTER JOIN** TableB
**ON** TableA.colmatch = TableB.colmatch




First return the TableA, if any records from TableB that match the TableA, return and fill the rest of blank with null.

**LEFT OUTER JOIN** with **WHERE**
Entries unique to TableA, those rows found in TableA and not found in TableB.

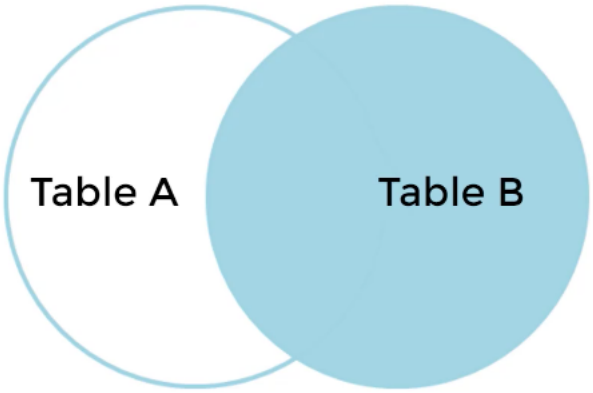
**SELECT** * **FROM** TableA
**LEFT OUTER JOIN** TableB
**ON** TableA.colmatch = TableB.colmatch
**WHERE** TableB.col1 **IS NULL**



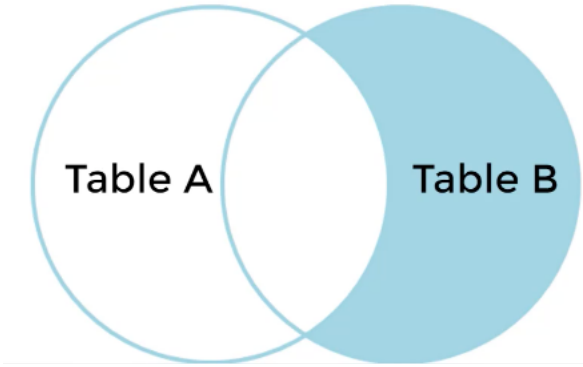

This returns records that TableA has but TableB doesn't.

**RIGHT OUTER JOIN**

```
SELECT * FROM TableA
RIGHT OUTER JOIN TableB
ON TableA.colmatch = TableB.colmatch
```




```
SELECT * FROM TableA
RIGHT OUTER JOIN TableB
ON TableA.colmatch = TableB.colmatch
WHERE TableA.col1 IS NULL
```



**UNIONS**

The UNIONS operator is used to combine the result-set of two or more SELECT statements. It basically serves to directly concatenate two results together, essentially “pasting” them together.

SELECT colname FROM TableA
UNION
SELECT colname FROM TableB

| Sales2021_Q1 | |
|---|---|
| name | amount |
| David | 100 |
| Claire | 50 |

| Sales2021_Q2 | |
|---|---|
| name | amount |
| David | 200 |
| Claire | 100 |

```
SELECT * FROM Sales2021_Q1
UNION
SELECT * FROM Sales2021_Q2;
```

| name | amount |
|---|---|
| David | 100 |
| Claire | 50 |
| David | 200 |
| Claire | 100 |

```
SELECT * FROM Sales2021_Q1
UNION
SELECT * FROM Sales2021_Q2
ORDER BY name;
```

| name | amount |
|---|---|
| David | 100 |
| David | 200 |
| Claire | 50 |
| Claire | 100 |

***JOIN Challenge Tasks***

- What are the emails of the customers who live in California?
  SELECT address.distrcit, customer.email
  FROM address
  INNER JOIN customer
  ON address.address_id = customer.address_id
  WHERE address.district = 'California'

| | district<br>character varying (20) | email<br>character varying (50) |
|---|---|---|
| 1 | California | patricia.johnson@sakilacust... |
| 2 | California | betty.white@sakilacustomer... |
| 3 | California | alice.stewart@sakilacustom... |
| 4 | California | rosa.reynolds@sakilacusto... |
| 5 | California | renee.lane@sakilacustomer.... |
| 6 | California | kristin.johnston@sakilacust... |
| 7 | California | cassandra.walters@sakilacu... |
| 8 | California | jacob.lance@sakilacustome... |
| 9 | California | rene.mcalister@sakilacusto... |

- Get a list of all the movies "Nick Wahlberg" (an actor) has been in.
  SELECT title, first_name, last_name
  FROM film_actor
  INNER JOIN  actor
  ON film_actor.actor_id = actor.actor_id
  INNER JOIN film
  ON film_actor.film_id = film.film_id
  WHERE first_name = 'Nick'
  AND last_name = 'Wahlberg'

| | title<br>character varying (255) | first_name<br>character varying (45) | last_name<br>character varying (45) |
|---|---|---|---|
| 1 | Adaptation Holes | Nick | Wahlberg |
| 2 | Apache Divine | Nick | Wahlberg |
| 3 | Baby Hall | Nick | Wahlberg |
| 4 | Bull Shawshank | Nick | Wahlberg |
| 5 | Chainsaw Uptown | Nick | Wahlberg |
| 6 | Chisum Behavior | Nick | Wahlberg |
| 7 | Destiny Saturday | Nick | Wahlberg |
| 8 | Dracula Crystal | Nick | Wahlberg |
| 9 | Fight Jawbreaker | Nick | Wahlberg |
| 10 | Flash Wars | Nick | Wahlberg |