Jose Portilla Udemy Course - The Complete SQL Bootcamp 2022: Go from Zero to Hero

My collection of personal notes from the course

Course Introduction

Overview

- Course will use PostgresSQL
- Databases and Tables Basics
- SQL Statement Fundamentals
- GROUP BY Clause
- Assesment Test over these topics
- JOIN
- Advanced SQL
- Commands
- 2nd Assessment test
- Create Databases and Tables
- Assessment Test 3

Extra lectures

- Views
- PostgreSQL with Python

Databases Overview

From Spreadsheets to Databases

Spreadsheets

- One time analysis
- quick charts

- small datasets
- untrained individuals

Databases

- Data Integrity (cant just click a cell and change it)
- massive data amounts
- quick combine different datasets
- automation
- backend for web apps and apps

Comparison

- Tabs in excel are like tables in a database
- rows and columns are the same

Database Platforms

- PostgreSQL = free & Open Source
- MySQL & MariaSQL = Free & Open Source, widely used on internet
- MS SQL Server Express = Free but has limitations, compatible with SQL Server (windows) only
- Microsoft access = Cost is bad -, not easy to use just SQL
- SQlite = Free (open source), mostly command line.

PostgreSQL is great for learning SQL

SQL (Structured Query Language) is the programming langauge used to communicate with databases

SQL is used in so much!

- MySQL
- PostGreSQL
- Oracle Databases
- MS Access
- Amazon Redshift
- Looker
- MemSQL
- Periscope Data
- Hive (runs on top of hadoop)
- BigQuery
- Facebook's Presto

All of these applications use SQL in the background

Most of the commands we learn are **not** specific to PostGreSQL

PostgreSQL = SQL Engine

PgAgmin - GUI Interface for PostgreSQL

SQL Statement Fundamentals

SELECT Statement

Most common used statement

Allows us to retreive data from tables

syntax

```
SELECT column_name FROM table_name;
```

You are **not** required to capitalize SQL commands but it is **best practice** as it seperates SQL Commands from tables, column names etc..

To grab all columns from the database, use:

```
SELECT * FROM table_name;
```

Word of caution: Using `SELECT *` should only be used when you need to utilize every column in the database. Calling this command greatly increases traffic between the database server and the application, which can affect the speed of other queries. AND HAS A LIST OF OTHER NEGATIVE EFFECTS

To get multiple columns,

```
SELECT column_1,column_2 FROM table_name;
```

Quick Note: the semicolon ; at the end of the query just signifies the end of the statement. This is **optional**, but similar to capitilazation, this makes your statements easier to read and is **best practice**

SELECT DISTINCT

Columns sometimes have duplicate values

Whenever you only want the distinct/unique values,

We use the `DISTINCT` keyword.

syntax,

```
SELECT DISTINCT column FROM table;
```

to clarify which column the distinct operator is being applied to,

```
SELECT DISTINCT(column) FROM table;
```

Parantheses are not necessary here but that does **not** apply to every SQL command

Note: when using `DISTINCT` with multiple columns and no parentheses, it gives you back distinct entries, so you may see the same value in one column more than once, but never the same entry in all columns for more than one row. It is a distinct collection of columns

```
SELECT DISTINCT column_1,column_2 FROM table_name;
```

Here is an example

COUNT Function

Returns the number of input rows that match a specific condition of a query.

COUNT(column_name) is a function so it **should have** the parentheses.

**IMPORTANT NOTE: ** `SELECT COUNT(column_name)` and `SELECT COUNT(*)` will return the same thing, since you are just returning the number of rows in the dataset which all have that column, so use

```
SELECT COUNT(*) FROM table_name
```

Note: The one instance where `SELECT COUNT(column_name)` is useful is for proper documentation. Say for instance you needed to count the number of some certain rows in table to answer a specific question about **one** column. When you look back on your `COUNT` function, the column name instead of * will help you remember the report you were trying to generate or question you were trying to answer.

`COUNT()` by itself clearly just returns the number of rows in a table, which is somewhat useful

However, when `COUNT()` is combined with other SQL commands like `DISTINCT`, it becomes much more useful.

imagine we want to know how many unique names are in a table?

```
SELECT COUNT(DISTINCT name) FROM table;
```

Side Note:

If we wanted to make the above query more readable, we could do:

```
SELECT COUNT(DISTINCT(name)) FROM table;
```

That is preferred IMO

SELECT WHERE Statement

SELECT and WHERE are the most fundamental SQL statements.

The **WHERE** statement allows us to specify certain conditions on columns for the rows that will be returned from our **SELECT** statement.

syntax

```
SELECT column_1, column_2
FROM table
WHERE conditions
```

Well what are these *conditions*?

Comparison Operators

Compare a Column value to something.

- Is the price greater than $3.00?
- Is the pet's name equal to "Sam"?

Operator	Description	Example
=	Equal to	(x=y) is not true
!=	Equal or not	(x!=y) is true
< >	Not equal to	(x<>y) is true
>	Greater than	(x>y) is not true
<	Less than	(x<="" td="">
>=	Greater than or equal to	(x>=y) is not true
<=	Less than or equal to	(x<=y) is true
!<	Not less than	(x!<="" td="">
!>	Not greater than	(x!>y) is true

Logical Operators

Allow us to combine multiple comparison operators

- AND
- OR
- NOT

Example,

```
SELECT name,choice
FROM table
WHERE name = 'David'
```

Now imagine multiple conditions

```
SELECT name,choice
FROM table
WHERE name='David' AND choice='Red'
```

Note that name and choice are representing two columns in a database

ORDER BY

Sort rows based on a column value

can be sorted numerically or alphabetically

syntax

```
SELECT column_1, column_2
FROM table
ORDER BY column_1 ASC/DESC
```

`ORDER BY` comes after selection and happens at the bottom of the query. It is the last thing SQL does.

If you leave the `ASC/DESC` part blank, then it defaults to `ASC`

You can use `ORDER BY` on multiple columns.

It will order by your first entry first and then take all of those ordered rows and order them by your second column entry.

LIMIT

Allows us to limit the number of rows returned for a query

Similar to `head()` in R or Python

Allows you to get a good grasp and preview of your returned data.

Also becomes very useful when paired with ORDER BY

Goes at the very end of a query, last command to be executed

syntax,

```
SELECT * FROM payment
ORDER BY payment_date DESC
LIMIT 10;
```

A SQL query like that could help us answer a helpful business question like, what are the 10 most recent payments we have had?

BETWEEN

Used to match a value against a range of values

- value `BETWEEN` low `AND` high

Using the `BETWEEN` operator is equal to using the >= **and** <= operators (it is inclusive).

You can also combine `BETWEEN` and the `NOT` operator to get `NOT BETWEEN`

This is the same as saying < low `OR` value > high
value `NOT BETWEEN` low `AND` high

We are **exclusive** of the actual low and high statements when using `NOT BETWEEN` unlike the regular expression

the `BETWEEN` operator can be used on dates. You need to format the dates in the **ISO 8601** standard format, which is `YYYY-MM-DD`

```
date BETWEEN '2007-01-01' AND '2007-02-01'
```

NOTE: datetime starts at 0:00, so pay close attention to inclusive and exclusive datetime queries

IN

Whenever you want to check if a value appears in some dataset

For example,

If you want to check if the name *adam* appears `in` some first_name column or users table.

syntax,

```
value IN(option1,option2, .... option_n)
```

examples,

```
SELECT color FROM table
WHERE color IN('red','blue')
```

Can be used with the `NOT` operator

```
SELECT color FROM table
WHERE color NOT IN('red','blue')
```

LIKE and ILIKE

We have already found solutions for queries where we need direct matches like first_name='john'

but what about certain situations where we need to query for a general pattern? Such as, emails ending in '@gmail.com' pr all names that begin with 'A'?

The `LIKE` operator allows us to permorm pattern matching against string data with the use of **wildcard** characters

What are **wildcard** characters?

1.) the **%** percent symbol, this allows us to match **any sequence of characters**
2.) the **_** underscore symbol, matches any **single** character

examples,

All names that begin with an 'A'

```
WHERE name LIKE 'A%'
```

Any string that starts with A and then has any sequence after that

All names that end with an 'a'

```
WHERE name LIKE '%a'
```

NOTE: `LIKE` is case-sensitive, we can use ILIKE which is case-insensitive.

using the underscore allows us to replace a single character

example,

image we want to get all mission impossible films

```
WHERE title LIKE 'Mission Impossible _'
```

this would give back mission impossible 1, mission impossible 2, etc...

You can use multiple underscoresm, which then leaves two character spots open,

We can combine pattern matching operators to create more complex patterns

```
WHERE name LIKE '_her%'
 - Cheryl
 - Theresa
 - Sherri

Has to be one character before 'her' but can have as many as you want after 'her'

**NOTE:** Keep in mind that PostgreSQL does support full regex capabilities.


good example,
```

SELECT * FROM customer
WHERE first_name LIKE 'A%' AND last_name NOT LIKE 'B%'
ORDER BY last_name

```
All customers where first name starts with an 'A' and last name does not start with 'B'
```

How to check if a given word is in the column entries?

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

This returns all the films with the word Truman in them.

GROUP BY Statements

Aggregate Functions

- AVG() -> Returns average value
- Count() -> Returns number of values
- MAX() -> Returns max value
- MIN() -> Returns min value
- SUM() -> Returns the sum of all values

Aggregate functions happen only in the **SELECT** or **HAVING** clause.

```
SELECT MAX(replacement_cost) FROM film
```

Keep in mind that adding another column

```
SELECT MAX(replacement_cost), title FROM film
```

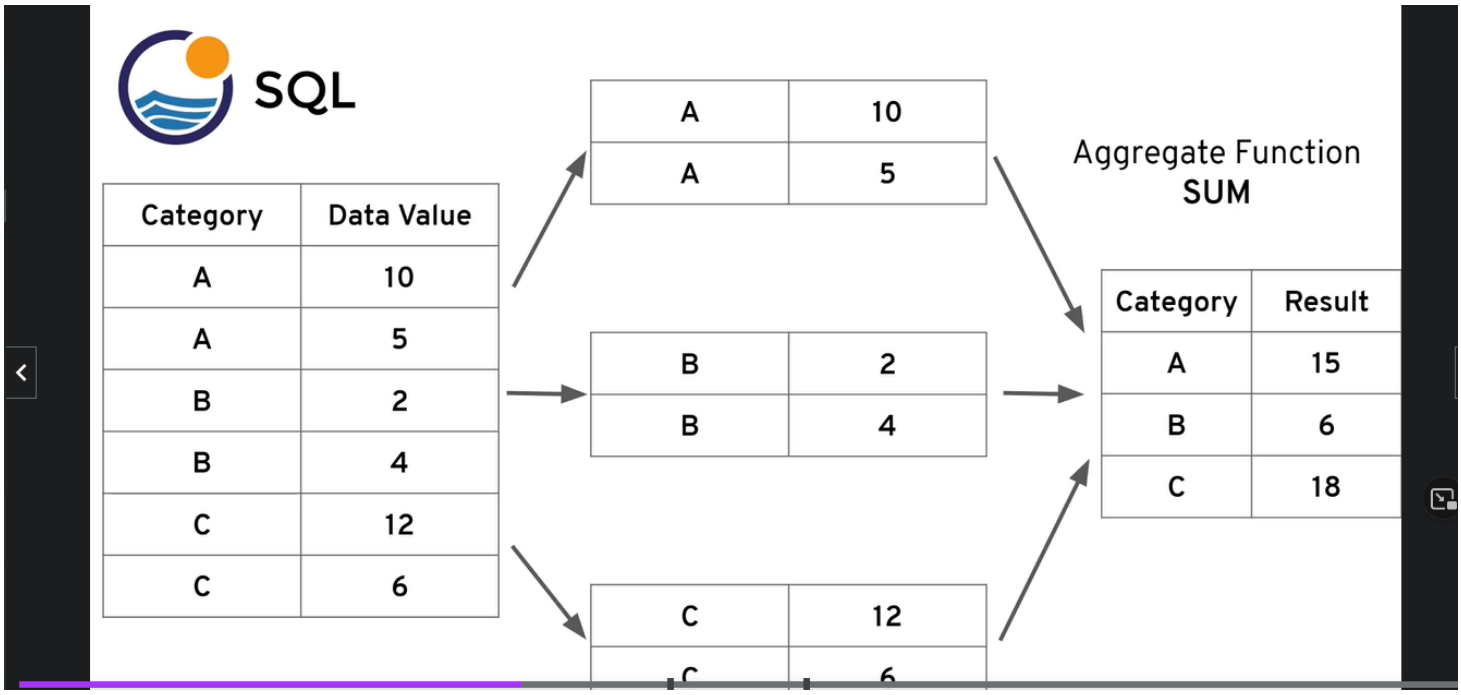
Would not work because our aggregate function is just returning a single value (in this case, max replacement cost)

Group By - Part 1

Allows us to aggregate columns per some category

Need to choose a **categorical** column to perform group by on

Categorical columns are non-continous

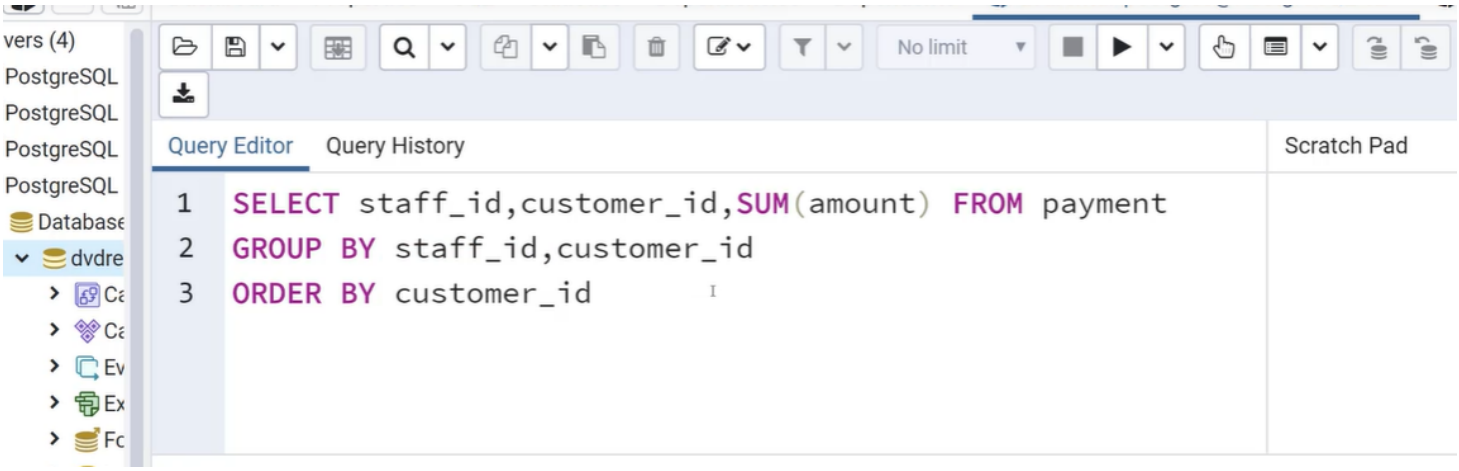



```
SELECT category_col, AGG(data_col)
FROM table
WHERE condition
GROUP BY category_col
```

The **GROUP BY** caluse has to come directly after a **FROM** statement or after a **WHERE** statement

In the **SELECT** statement, columns must either have an aggregate function or be in the **GROUP BY** call

Group By - Part 2



Some important notes,

- Order of the columns in the `SELECT` clause does not matter
- Order of the columns in the `GROUP BY` clause does not matter?
- Here, you can order by any columns or all columns

HAVING Clause

Allows us to filter after an aggregation has already taken place

We can not use `WHERE` to filter by aggregate results because those happen after a `WHERE` is executed

`HAVING` allows us to use the aggregate result asa filter along with a `GROUP BY`

EXAMPLE:

Give us customer id's of customers with over 40 transactions

```
select count(transactions), customer_id
from transactions
group by customer_id
HAVING count(transactions) >= 40
```

JOINS

`JOINS` allow us to combine data from multiple tables

The reason that there are so many types is because we have to decide what to do with data present in only one of the tables

AS Clause

Allows us to create an alias for a column or result

How to rename a column in SQL result,

```
SELECT column as new_name
FROM table
```

How to rename result,

```
SELECT SUM(column) AS new_name
FROM table
```

How is this useful?

Well if we are demonstrating analytics and want to rename a column,

```
SELECT SUM(amount) AS net_revenue
FROM payment;
```

NOTE: The `AS` operator gets executed at the very end of a query, meaning that we can not use the ALIAS inside of a `WHERE` operator

Basically you can not use your alias inside your sql statements

Another useful alias,

```
SELECT COUNT(*) AS num_transactions
FROM payment
```

INNER JOIN

The `INNER JOIN` is the most basic join operation

the result of this `INNER JOIN` is the set of records that **match in both** tables

syntax,

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

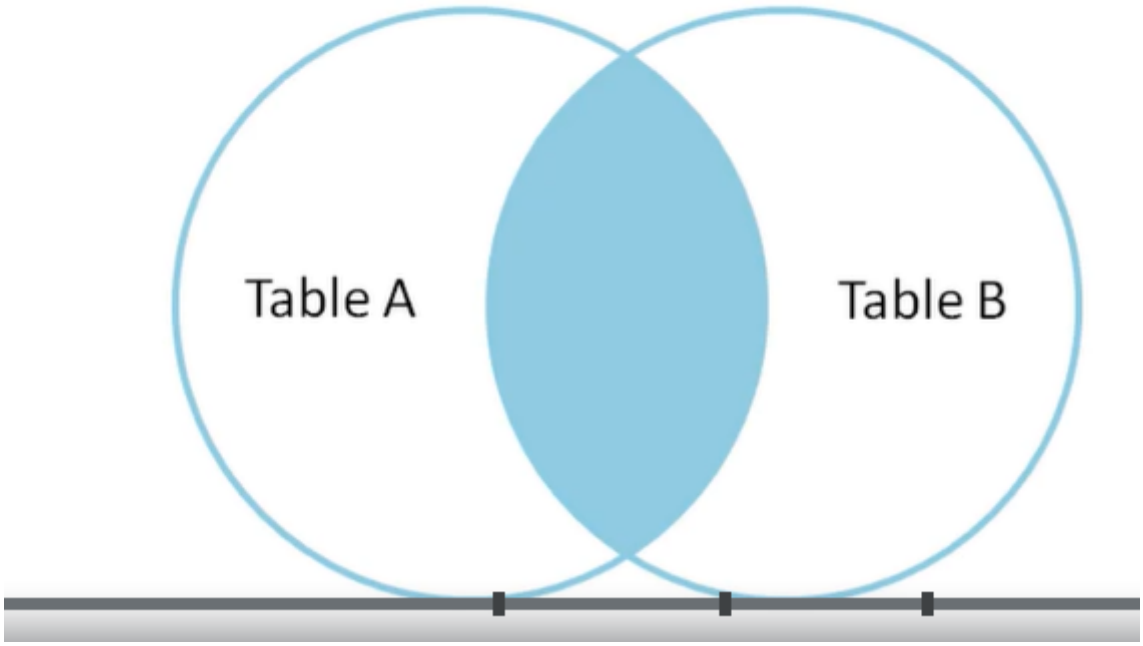
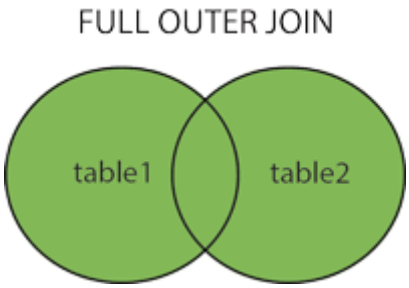


Table order does not matter in the `INNER JOIN`

In postgresSQL, you can just use `JOIN` for an inner join

FULL OUTER JOIN

OUTER JOINS allow us to specify how to deal with values only present in one of the tables being joined

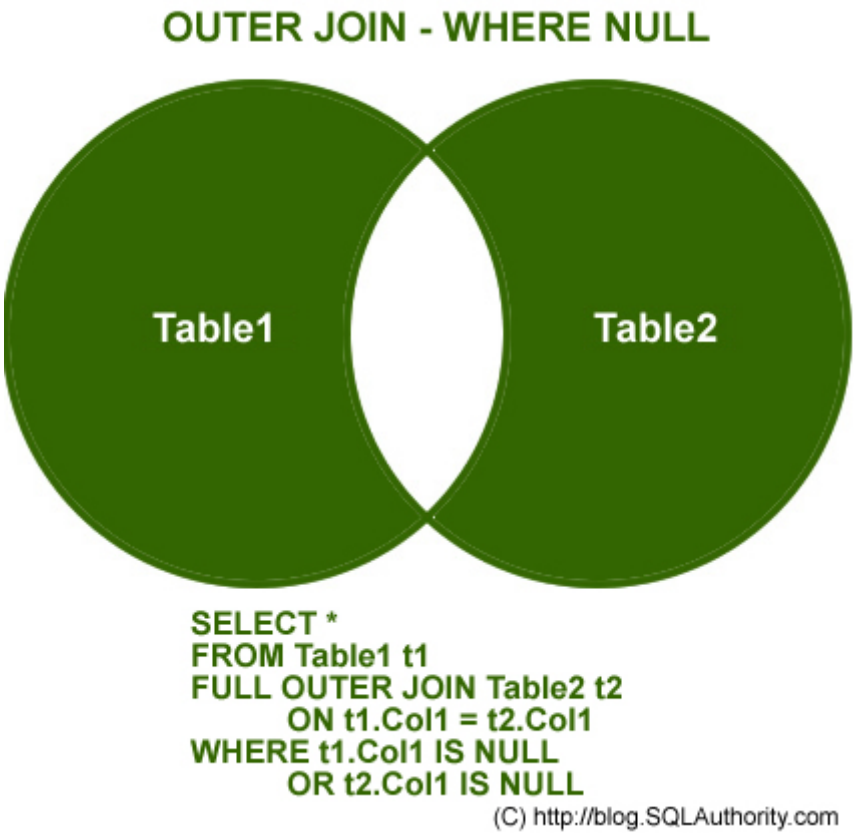


Syntax,

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

A **FULL OUTER JOIN** grabs everything from both tables and *attempts* to match it with the other table, however if it fails to match then it will still add all records regardless (which then creates a bunch of null values)

There is also the option to just get the nonmatching records from both tables, we simply use a **WHERE** clause and identify records that returned a null value



LEFT OUTER JOIN

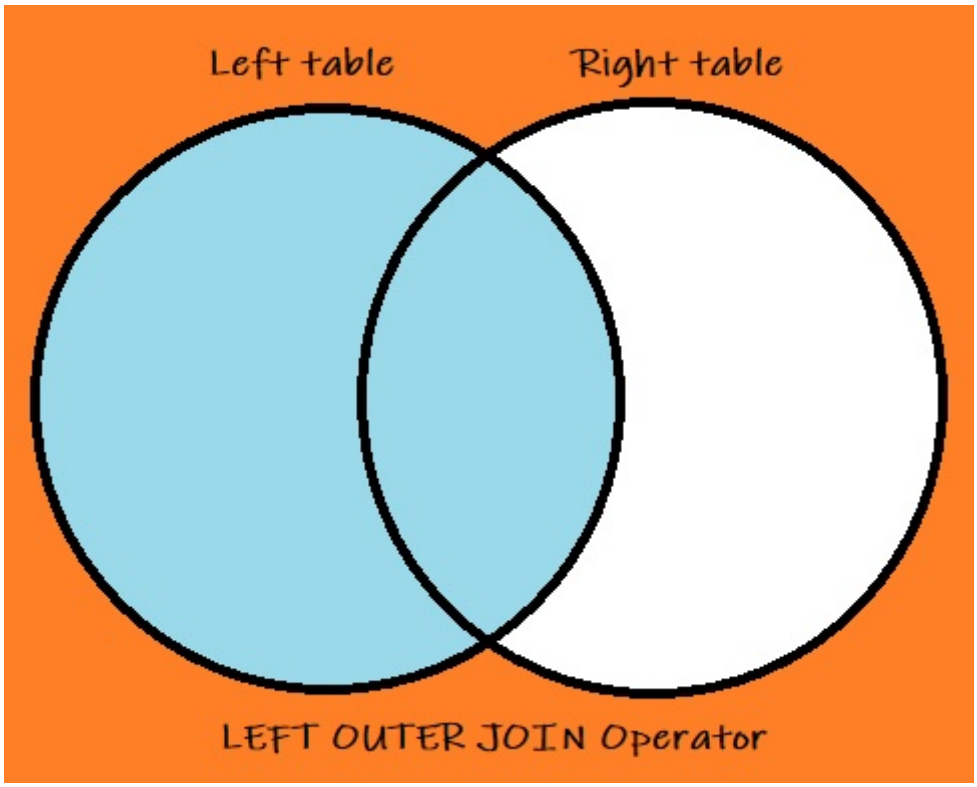
A **LEFT OUTER JOIN** results in the set of records that are in the left table, if there is no match with the right table, the results are null.

We can add the **WHERE** statement to further modify our **LEFT OUTER JOIN**

Syntax,

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

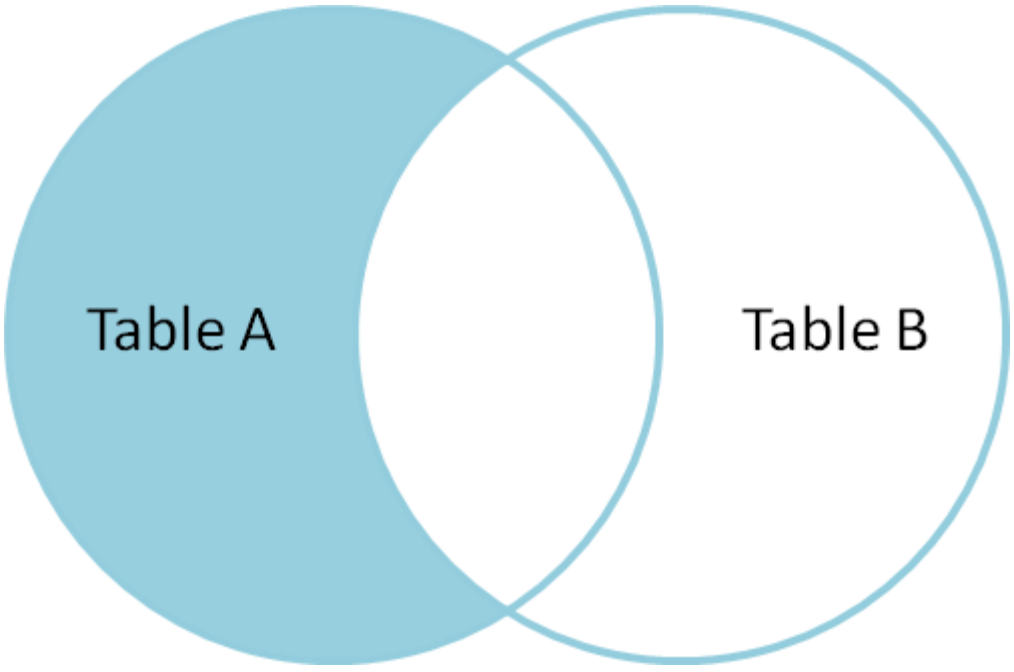
NOTE: Order matters! This is not a symetrical operation



What if we only wanted entries that our unique to our left table and not matching ones with the right table?

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

That statement gives us the following,



RIGHT JOIN

A **RIGHT JOIN** is essentially the same as a **LEFT JOIN**, except the tables are switched.

This would be the same as switching the table order in a **LEFT OUTER JOIN**

Syntax,

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

By just using the left join, it can simplify your mental space (all preference)

This section was skipped a bit due to the point above

UNION

- The **UNION** 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.

Syntax,

```
SELECT column_name(s) FROM table1
UNION
SELECT column_name(s) FROM table2
```

Advanced SQL Commands

Timestamps and Extract - Part One

PostgreSQL can hold date and time information

- **TIME** - Contains only time
- **DATE** - Contains only date
- **TIMESTAMP** - Contains date and time
- **TIMESTAMPZ** - Contains date, time, and timezone

When creating a database, careful consideration should be taken into account when deciding time types

Here are some functions that are useful when populating a database,

- **TIMEZONE** - Gives you your current timezone
- **NOW** - Gives you *TIMESTAMPZ* information right now
- **TIMEOFDAY** - A text string value of NOW
- **CURRENT_TIME** - Subsets of the commands above, gives current time with timezone
- **CURRENT_DATE** - gives current date yyyy-mm-dd

Timestamps and Extract - Part Two

EXTRACT()

Allows you to **"extract"** or obtain a sub-component of a date value

- Year
- Month
- Day
- Week
- Quarter

Syntax,

```
EXTRACT(YEAR FROM date_col)
```

AGE()

Calculates and returns the current age given a timestamp

syntax,

```
AGE(date_col)
```

This returns,

```
13 years 1 mon 5 dats 01:34:13.003423
```

TO_CHAR()

General function to convert data types to text

Useful for timestamp formatting

syntax,

```
TO_CHAR(date_col,'mm-dd-yyyy')
```

Mathematical Functions and operators

There is a vast majority of mathematical operators to utilize in SQL

There is loads of documentation on all of them according to your DB

- Addition
- Subtraction
- Multiplication
- Division
- Modulus

String Functions and Operators

Again, there is a vast majority of string function that we can utilize in SQL

Here are some of the more useful string functions,

- ASCII()
- BIN()
- BIT_LENGTH()
- CHAR_LENGTH()
- CONCAT()

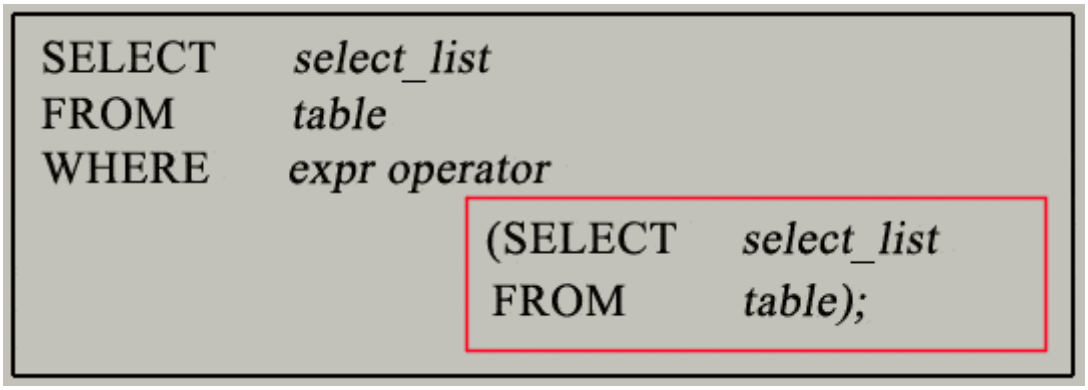
Here is some good documentation for string functions

SubQuery

Helps us get the result in a single query request

- Compare an expression to the result of the query
- Determine if an expression is included in the results of the query
- Check whether the query selects any rows
- The subquery (inner query) executes once before the main query (outer query) executes.
- The main query (outer query) use the subquery result.

Syntax,



Example,

```
SELECT student, grade
FROM test_scores
WHERE student IN
(SELECT student
FROM honor_roll_table)
```

A useful operator to use with the SubQuery is the `EXISTS` operator

The `EXISTS` operator returns true or false based on if there are any records returned

```
SELECT column_name
FROM table_name
WHERE EXISTS
(SELECT column_name FROM
table_name WHERE condition);
```