SQL

INDEX

- CREATION & MODIFYING TABLES

  THEORETICAL CONCEPTS
  - DATATYPE
  - PRIMARY KEY & FOREIGN KEY
  - CONSTRAINT

  DATA MODIFY Queries

  - CREATE
  - INSERT
  - UPDATE
  - DELETE
  - DROP/TRUNCATE
  - ALTER

CLAUSES

- ⇨ **SELECT . . . FROM . . .**
  - **DISTINCT**
  - **COUNT**
- ⇨ **WHERE . . .**
- ⇨ **ORDER BY . . . LIMIT . . .**
  - **ORDER BY Extension**
    FETCH
    OFFSET
- ⇨ **GROUP BY . . . HAVING . . .**
  - **GROUP BY Extensions**
    GROUING SETS
    ROLLUP
    CUBE

```
/* --- 1. CLAUSES --- */
- SELECT ... FROM ...
- SELECT 'EXTENSIONS' {"DISTINCT", "COUNT"}
- WHERE ...
- ORDER BY ... LIMIT ...
- ORDER BY 'EXTENSIONS' {"OFFSET", "FETCH", "WITH TIES"}
- GROUP BY ... HAVING ...
- GROUP BY 'EXTENSIONS' {"GROUPING SET", "ROLLUP", "CUBE"}
```

SELECT … FROM …

- **SELECT** is used to **retrieve information** from the Table. i.e., 'Column' or 'Attributes' from the Dataset.
- "*****" is used to get all the columns in a Table.
- While specific columns separated with comma can be mentioned to retrieve specific column.
- **SELECT … FROM** is the main Query Statement. We can say it is must for any of the SQL statement.
- It forms the base of SQL statement.
- **FROM** decides from which Table or the Dataset we want to retrieve the Columns.
- Syntax:- **SELECT column_name(s) FROM table_name;**

```
SELECT * FROM actor;
SELECT first_name,last_name FROM actor;
```

➢ DISTINCT

- Many a times Column contains '**Duplicate Values**' or table contains duplicate rows. If we want to retrieve only distinct values from a column or a set of columns, we use keyword **DISTINCT**.
- It is used in SELECT Statement before the column name to mention on which column we have to apply this function.
- Syntax :- SELECT **DISTINCT column_name** FROM table_name; SELECT **DISTINCT(column_name)** FROM table_name;

```
SELECT DISTINCT rating FROM film;
SELECT DISTINCT(rating) FROM film;
```

➢ COUNT

- **COUNT** returns the number of the records that matches the conditions in a column.
- We can apply COUNT on specific column or on the whole table. That is COUNT(*) or COUNT(column_name).
- COUNT is a column function. And it is used where the column_name is listed in the SELECT statement.
- Syntax :- SELECT **COUNT(column_name)** FROM table_name;

```
SELECT COUNT(*) FROM film;
SELECT COUNT(title) FROM film;
SELECT COUNT(DISTINCT rating) FROM film;
```

WHERE

Filtering Records is one of the **special functionality** in the Data Analytics. Many a times, we need to filter the table based on certain criteria. WHERE is the clause in standard SQL which allows us to do this.

- WHERE is another most important Function in SQL which makes part of SELECT statement.
- **WHERE** apply the **filtering condition** on the data records or Rows and specify which rows to be retrieved i.e., on which basis we have to filter the data and retrieve it.
- We can apply the **Arithmetic Operator** for comparing values. Like ">", "<", "=", "!="
- We can **conjunct** different condition using **Logical Operator** AND, OR, or apply negation condition NOT.
- We can even compare it with the scalar result of Subquery. We'll see that later.
- It is places at the end of SELECT Clause.
- Syntax :- SELECT column_name FROM table_name **WHERE condition(s)**;

```
SELECT * FROM film
WHERE rating = 'R' AND replacement_cost >= 19.99 OR replacement_cost <= 10;
SELECT email FROM customer WHERE first_name = 'Nancy' AND last_name = 'Thomas';
SELECT phone FROM address WHERE address = '259 Ipoh Drive';
```

ORDER BY

- **ORDER BY** operator is used to "**Sort**" the resultant output data based on column(s) value.
- We can also mention **multiple column** in specific order to sort the rows based on multiplt columns. In this way, it will sort first based on the first mentioned column and subsequently sort later column.
- Sorting can be done either Alphabetically or Numerically based on the column type.
- **ASC** and **DESC** can be used AFTER mentioning each column_name to specify whether we have to sort in ascending or Descending order for that column. Default is ASC.
- ORDER BY comes after SELECT Statement.
- Syntax :- SELECT FROM WHERE **ORDER BY column_name(s) ASC/DESC LIMIT n**;

```
SELECT store_id, first_name, last_name FROM customer
ORDER BY first_name DESC;

SELECT store_id, first_name, last_name FROM customer
ORDER BY store_id, first_name;

SELECT store_id, first_name, last_name FROM customer
ORDER BY store_id DESC, first_name ASC;
```

➢ LIMIT

- **LIMIT** is used to set hard limit of **number of records** we want to retrieve.
- It is useful for not returning every single row, but only view the top few rows to get an idea of table layout.
- It is pretty useful in combination with ORDER BY after sorting.

```
SELECT * FROM payment
WHERE amount != 0.00
ORDER BY payment_date LIMIT 5;

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

➢ ORDER BY Extensions

➢ OFFSET

- **Skip first n_rows** for retrieving.
- Can only be placed at an end of statement.
- Can be used either 'Separately' or as an 'Extension of ORDER BY'
- Syntax := **OFFSET n_rows**.

```
SELECT store_id, first_name, last_name FROM customer
OFFSET 5;

SELECT store_id, first_name, last_name FROM customer
ORDER BY store_id, first_name OFFSET 5;
```

➢ FETCH

- **Retrieve next n_rows** only.
- Can only be placed at an end of statement.
- Can be used Either 'Separately' or as an 'Extension of ORDER BY'.
- Syntax := **FETCH FIRST n_rows ROWS ONLY**;

```
SELECT store_id, first_name, last_name FROM customer
FETCH FIRST 10 ROWS ONLY;

SELECT store_id, first_name, last_name FROM customer
ORDER BY store_id, first_name FETCH FIRST 10 ROWS ONLY;

SELECT store_id, first_name, last_name FROM customer
ORDER BY store_id, first_name OFFSET 5 FETCH FIRST 10 ROWS ONLY;
```

GROUP BY

Grouping is a **special functionality** in data analytics. Grouping allows to group multiple records and return an aggregated result on a specific column. Aggregated function can be MAX, MIN, SUM, AVG, COUNT and many more. Many a times, we have to group records based on categories from categorical column. Like group records for each of the month, or group records based on the region, etc. GROUP BY is a function for such purpose in standard SQL.

- **GROUP BY** performs the aggregation function on column, based on per "Categorical Classes" of the grouping column.
- Remember that **Aggregation Function** is must for GROUP BY Clause.
- If we want to sort based on the aggregation function result, make sure to mention the whole aggregation function in the ORDER BY Clause.
- GROUP BY clause must be placed after the SELECT FROM WHERE Statement. And Before Order By if Sorting.
- Syntax :- SELECT **col_name_for_grouping, AGG_FUN(col_name_for_agg_fun)** FROM table_name **GROUP BY col_name_for_grouping;**

Important Note --> We can only SELECT those columns for retrieving which either we have based our grouping on AND those columns on which we are applying our aggregation fun.

```
SELECT staff_id,customer_id, SUM(amount) FROM payment
GROUP BY staff_id,customer_id
ORDER BY customer_id
LIMIT 10;

SELECT DATE(payment_date), SUM(amount) FROM payment
GROUP BY DATE(payment_date)
ORDER BY SUM(AMOUNT) DESC
LIMIT 5;
```

Important Notes regarding GROUP BY with examples below

- Example 1, We can choose to display/SELECT all the grouped column in the SELECT statement.
- Example 2, We can choose to NOT SELECT any grouped column as not to display them. But it doesn't make sense as it isn't understandable or comprehensible. It should be done based on context.
- Example 3, we CANNOT select those columns which we are not using as either grouping or for aggregating, just to retrieve/display it. Because it is grouped in the grouping column so logically it is not possible for it to be displayed individually. For E.g. - Take a look at below example, in this as we have grouped customer_id, we cannot SELECT 'staff_id' to retrieve as both Staff_id_1 & Staff_id_2 are grouped for any customer so logically it is not possible to display them individually.

```
-- Example 1 -- -- SELECT all the GROUPED column. usually what we should do. --
SELECT staff_id,customer_id, SUM(amount) FROM payment
GROUP BY staff_id,customer_id
ORDER BY customer_id
LIMIT 10;

-- Example 2 -- -- Choose to display either staff_id or customer_id based on context. --
SELECT customer_id, SUM(amount) FROM payment
GROUP BY staff_id,customer_id
ORDER BY customer_id
LIMIT 10;

-- Example 3 -- -- We cannot SELECT other column which is not the Grouping column. --
--     i.e., SELECTED columns must either be in GROUP BY clause or in aggregation function. --
SELECT customer_id, SUM(amount) FROM payment
GROUP BY customer_id
ORDER BY customer_id
LIMIT 10;
```

➢ HAVING

- **Filtering Condition**, same as WHERE Clause, but **ON aggregated functions results** which has already taken place.
- Placed after the GROUP BY function in GROUP BY CLAUSE.
- Syntax :- SELECT FROM WHERE ... GROUP BY ... HAVING **condition_on_agg_fun;**

```
SELECT customer_id, SUM(amount), COUNT(amount) FROM payment
WHERE amount<5
GROUP BY customer_id
HAVING count(amount)<35      --Same as WHERE clause but for Aggregated Function columns
ORDER BY SUM(amount) DESC
LIMIT 10;
```

➢ GROUP BY Extension

Important Note –

Below are the three extensions of GROUP BY for aggregating based on multiple possible combinations of Grouped Column(s). All these Extensions allows us to perform Aggregation Function on ALL the "Possible Categories Combination" of either some or all the "Possible Combination of SUBSETS from within the SUPER SET of Grouped Column(s)" rather than just the single one as in the case of simple GROUP BY.

These are very useful Extensions for GROUP BY to simplify Complex Query and increase efficiency. Remember, those Column which are not used for aggregating in particular grouping/partitions, will be mention as "NULL" in the Data Output. So it's in the best practice to Replace it with some understandable text like "TOTAL" with COALESCE.

➢ GROUPING SETS

- GROUPING SETS allows us to group any of the **'POSSIBLE COLUMN COMBINATION(s)**' from our **'MULTIPLE GROUPING COLUMN**'.
- We can give our **'customize set of combinations'** from GROUPING COLUMN(S) to execute grouping call.
- E.g. Let us say, we are performing GROUP BY on grouping set (c1, c2, c3); With standard GROUP BY, we can perform GROUP BY on this "particular combination" only. But if we want to perform GROUP BY on different combination say (c1, c2), (c3, c2), (c1), (c3), (), we need to UNION ALL this tables as a separate query. But that'll not be efficient. Instead, we can use **GROUPING SETS** Keyword
- Look example for GROUP BY of (c1, c2) and GROUP BY of (c1, c2, c3) here - Syntax :=
- **GROUP BY GROUING SET ((c1, c2), (c1, -), (-, c2), (-, -));**
- **GROUP BY GROUING SET ((c1, c2, c3), (c1, c2, -), (c1, c3, -), (-, c2, c3), (c1, -, -), (-, c2, -), (-, -, c3), (-, -, -));**
  (Note – 1. I've mentioned all the combination just for understanding purpose. We can perform group by on the subset of this. 2. I've used "-" for understanding purpose. It is not needed in the actual syntax.) Only mention those column_order on which want to partition for grouping.
- In general, we can select from within the 2^n combinations for n mentioned column in Input Column(s).
- To retrieve all the possible combination, we can directly use CUBE (Refer lesson for CUBE after ROLLUP).

```
--# First, We will run without GROUPING SET
(SELECT staff_id, NULL, category_name, COUNT(*) FROM staff_film
GROUP BY staff_id, category_name ORDER BY staff_id)
UNION ALL
(SELECT staff_id, rating, NULL, COUNT(*) FROM staff_film
GROUP BY staff_id, rating ORDER BY staff_id)
UNION ALL
(SELECT NULL, rating, NULL, COUNT(*) FROM staff_film
GROUP BY rating ORDER BY rating)
UNION ALL
(SELECT NULL, NULL, category_name, COUNT(*) FROM staff_film
GROUP BY category_name ORDER BY category_name);
```

```
--# Now, look at the convenience of above query with 'GROUPING SET'
SELECT staff_id, rating, category_name, COUNT(*)
FROM staff_film
GROUP BY GROUPING SETS ( (staff_id, category_name), (staff_id, rating), (rating), (category_name))
ORDER BY staff_id, rating, category_name;
--# Same result but with much more simple Query AND Efficient Underlying Execution.
```

➢ ROLLUP

- It allows us to perform Aggregation Function on all the "Possible **Hierarchical Combinations**" moving from Bottom Most to Top Order of Input Column(s) | Inner most to Outer Order. (c1, c2, c3), (c1, c2, -), (c1, -, -)
- Meaning, if we are ROLLING UP on multiple columns, ROLLUP will super aggregate moving from bottom most hierarchy to uppermost.
- That is to say
  First, it results in an aggregation of all the outermost partitions grouped together.

Second, it results in one more row of result of 'super aggregated calculation' of its sub-partitions.
Lastly, it results in one more row for result of 'super-super aggregated calculation' of above aggregation.

- ROLLUP considers a hierarchy on which order the column is mentioned. (c1, c2, c3) --> c1 > c2 > c3.
- Few Examples with 1 or 2 or 3 Column ROLLUP. Syntax :=
- GROUP BY ROLLUP(c1); --> {(c1), (c-)}
- GROUP BY ROLLUP(c1, c2); --> {(c1, c2), (c1, -), (-, -)}
- GROUP BY ROLLUP(c1, c2, c3); --> {(c1, c2, c3), (c1, c2, -), (c1, -, -), (-, -, -)}
- GROUP BY c1, ROLLUP (c2, c3); --> {(c1, c2, c3), (c1, c2, -), (c1, -, -)} --# Partial ROLLUP.
- As ROLLUP consider a hierarchy of column, so **remember** - Specified Column Order in Input Column(s) for this function call is important as it creates Hierarchy based on it.
- It is basically in literal terms UNIONing grouping of innermost to outermost hierarchy of mentioned order of set of columns.

```
SELECT staff_id, rating, category_name, COUNT(*)
FROM staff_film
GROUP BY ROLLUP (staff_id, rating, category_name)
ORDER BY staff_id, rating, category_name;
--# Look at how the different combination of aggregation from innermost to outermost hierarchy is returned.
--# staff_id, rating, name --> staff_id, rating, NULL ---> staff_id, NULL, NULL --> NULL, NULL, NULL.
```

➢ CUBE

- CUBE is special case of GROUP BY's GROUPING SET. By default, it selects "ALL" the 'possible **2^n combination**' in SUPER SET of Grouping Column(s), as mentioned above. Syntax :=
- GROUP BY CUBE(c1, c2, c3);
- GROUP BY c1, CUBE(c2, c3); --# Partial CUBE.
- It is a Shorthand for GROUPING SETS to select all combination.

```
SELECT staff_id, rating, category_name, COUNT(*)
FROM staff_film
GROUP BY CUBE (staff_id, rating, category_name)
ORDER BY staff_id, rating, category_name;
```

```
SELECT staff_id, rating, category_name, COUNT(*)
FROM staff_film
GROUP BY staff_id, CUBE (rating, category_name)
ORDER BY staff_id, rating, category_name;
```

CLASUE OVERVIEW

CLAUSES are the main skeleton or we can say main component for the query statement. All the keyword such as operators or function or joins forming the complex query will be part of these clauses only. Among these, SELECT and FROM are the compulsory clauses to retrieve data as they specify which data to retrieve and from where to retrieve. Optional Clause WHERE specify the filtering condition for rows. Optional Clause GROUP BY specifies whether we have to group per category. In the end Optional ORDER BY allows us to order or sort the retrieved data from these clauses. Order of these clauses in the Query Statement.

SELECT … FROM … WHERE… GROUP BY … HAVING … ORDER BY … LIMIT …;

⇨ SELECT = It Retrieves the Data from the Table for Selected columns. We can select columns by modifying it through FUNCTIONS also. Columns sets or modified columns are going to be part of this clause. DISTINCT =

Distinct is used to retrieve Distinct records using the input columns. COUNT = Count is used to count the number of records retrieved either on column or on whole Table.

- ⇨ FROM = It is used to mention the TABLE from which we are referring the data. We can retrieve data either from Single Table or from Joined Table or from Temporary Table. In short, table are to mentioned in this clause
- ⇨ WHERE = It is used to mention the conditions on which basis we have to filter the row. Basically, it is Filtering conditions for records to be retrieved. Filtering condition using OPERATORS are to mentioned in this clause.
- ⇨ ORDER BY = Order by is used to 'sort' the retrieved data based on specific column or combination of columns. ORDER BY extension LIMIT = LIMIT is used to mention how many rows we have to fetch after sorting ascendingly or descending. OFFSET = Offset is used to skip first mentioned number of rows from the resultant data of the query. FETCH = Fetch is used to fetch next n number of rows after skipping fixed number of rows.
- ⇨ GROUP BY = Group by is used to group column values, per category from another column, using some aggregation function. The resultant data is the aggregated measure that we have selected on each of the category or partitions. GROUP BY extension HAVING = HAVING is used as same as WHERE but to apply the filtering condition on the aggregated result. GROUPING SETS = Grouping sets allows us to select multiple combination of columns from the grouping column set to aggregate data based on different partition criteria. CUBE = Cube is the special case of Grouping sets in which all the aggregation is executed on all the possible combination of column from the grouping column set. ROLLUP = ROLLUP performs the aggregation function on column combinations moving from the innermost hierarchy to outermost hierarchy from the grouping column set.

OPERATOR


```
/* --- 2. OPERATOR --- */
- ARITHMETIC OPERATOR
- COMPARISON OPERATOR
- LOGICAL OPERATOR
- FILTERING OPERATORS
- SPECIAL OPERATOR {"ALIAS", "CONCATENATION", "WILDCARD"}
```

Arithmetic Operator

- There are 5 Arithmetic operator in SQL. SQL also follow **BODMAS rule** for complex formulas.
- **"+", "-", "*", "/", "%"**, "BODMAS"
- Used to Perform Arithmetic Operations on Columns, whether in the SELECT or in the WHERE.

Comparison Operator

- There are 6 Comparison Operator in SQL.
- **"=", "!= | <>", ">", "<", ">=", "<="**
- Used to compare results in WHERE statement conditions.

Logical Operator

- **"AND", "OR", "NOT"**
- Used to perform Logical Operators on the Conditions

Filtering Operator

- **"IS NULL", "BETWEEN", "IN", "LIKE/ILIKE", "ALL/ANY", "EXISTS"**
- Very Important Note = All these special operators can be used with conjunction of Logical Operator "**NOT**". i.e., NOT IN, NOT BETWEEN, NOT LIKE, NOT EXISTS, NOT ALL/NOT ANY, IS NOT NULL.

IS NULL

- “IS” is special Filtering Operator which can only be used with NULL predicate.
- Comparison Operator with NULL results in not True or False, but in NULL or Unknown.
- So "**IS NULL**" allows us to compare if the value is NULL
- Syntax := IS NULL, IS NOT NULL

BETWEEN

- **BETWEEN** operator is used to match a column values against a **range of values**, in filtering condition.
- Syntax :- **WHERE column_name BETWEEN low AND high**.
- It automatically conjunct two filtering condition - Value greater than LOW AND less than HIGH **both Inclusive**.
- Can also be used as **NOT BETWEEN** low AND high. - Value less than LOW AND greater then HIGH both Exclusive.
- Can also be **used with dates** datatype. Dates should be in ISO 8601 or American format of YYYY-MM-DD.

```
SELECT COUNT(*) FROM payment
WHERE amount NOT BETWEEN 8 and 9;

SELECT * FROM payment
WHERE payment_date BETWEEN '200-02-01' AND '2007-02-15';
/* Give Special note to inclusivity for the last date as PostgreSQL consider date starts form 00:00 hour. */
```

IN

- Sometimes we have to check multi possible option for values in column. Its redundancy to use OR again and again.
- We can use **IN operator** to create a condition that checks if the values is 'IN' or 'match to' any of the options included in the **list**.
- It conjunct all the filtering conditions such as - value = 'A1' OR value = 'A2' OR value = 'A3' and so on....
- It is used in Filtering Condition as a list. Syntax :- **WHERE column_name IN ('A1','A2','A3',...)**
- We can also use **NOT IN** to specify that the value should not match to any of the options from the set.

```
SELECT * FROM customer
WHERE first_name IN ('John','Jake','julie');

SELECT * FROM payment
WHERE amount NOT IN (0.99,1.98,1.99);
```

LIKE

- What if we want to match the value against the general **pattern of a string**. We can use **LIKE** operator to perform such functionality.
- LIKE uses **wildcard characters** to match against a general pattern as specified.
- **"%"** = Match against any number/sequence of characters (0 or more). **"_"** = Match against any single character.
- LIKE is case sensitive. **ILIKE** is case insensitive.
- It is like Regex in normal Programming language. And Postgre support full REGEX capabilities: https://www.postgresql.org/docs/12/function-matching.html
- Syntax = **WHERE column_name LIKE 'Pattern'**

```
SELECT COUNT(*) FROM customer
WHERE first_name LIKE 'J%' AND last_name LIKE '%e'; -- Name that start with "J" and ends with an "e".

SELECT first_name FROM customer
WHERE first_name NOT LIKE '%er%'; -- Name that not has an 'er' anywhere in the string.

SELECT first_name FROM customer
WHERE first_name LIKE '_her%'; -- Name that has 'her' at 2nd position.
```

ALL

- **ALL** retrieve Data if it **satisfies** the **'Comparing Condition'** with **"ALL" the 'SET VALUES** returned by Subquery'.
- It is **succeeded by SUBQUERY or a LIST**.
- ALL Operator compare the field_value with all the values from the given 'SET'; AND, retrieve it only when it satisfies the specified condition with "EACH" value from the SET.
- **'SET'** --> (Either Mostly from "Column Output of Subquery Result" OR "explicit list").

- It must be preceded by **comparison_operator** for 'Comparing Condition'.
- It can be used in both the WHERE or HAVING clause
- Syntax := **WHERE field_value any_comparison_operator ALL (SET from Subquery);**

```
/* --- Retrieve those Customer IDs whose Maximum payment amount among all their payments is NOT greater than
       Average amount of ALL the Customer's ID. --- */
SELECT customer_id, MAX(amount) AS max, MIN(amount) AS min, AVG(amount) AS avg, SUM(amount) AS sum
FROM payment GROUP BY customer_id
EXCEPT
SELECT customer_id, MAX(amount) AS max, MIN(amount) AS min, AVG(amount) AS avg, SUM(amount) AS sum
FROM payment GROUP BY customer_id
HAVING MAX(amount) >= ALL (SELECT AVG(amount) FROM payment GROUP BY customer_id)
```

ANY

- **ANY** retrieves data if it **satisfies** the **'Comparing Condition'** with **"ANY" of the 'SET VALUES** returned by subquery'.
- It is **succeeded by SUBQUERY or a LIST**.
- ANY Operator compare the conditional field value with all the values from the given 'SET'; AND, retrieve it only when it satisfies the condition with "ANY" value from the SET.
- **SET** --> (Either Mostly from "Column Output of Subquery Result" OR "explicit list").
- It must be preceded by **comparison_operator** for 'Comparing Condition'.
- It can be used in both the WHERE or HAVING clause
- Syntax := **WHERE field_value any_comparison_operator ANY (SET from Subquery);**
- "SOME" KEYWORD use case is same that of "ANY". they can use interchangeably.

```
/* -Retrieve those customers whose MAX payment is less than "ANY" of the customer's Average Payment Amount - */
SELECT customer_id, MAX(amount) AS max, MIN(amount) AS min, AVG(amount) AS avg, SUM(amount) AS sum
FROM payment GROUP BY customer_id
HAVING MAX(amount) <= ANY (SELECT AVG(amount) FROM payment GROUP BY customer_id)
```

EXISTS

- **EXISTS** keyword check for **Existence of Rows or Records of Main query within the Subquery**.
- It is **succeeded by SUBQUERY**.
- It retrieves the record only if it is present in subquery. that is 'T' in subquery.
- Very Important point to remember here, for EXISTS we **do not have to check ALL the columns** of main query within the Subquery. We can check for exist**ence of Row within the main query using ANY of the column(attribute) in Subquery**.
- Syntax := **<MAIN QUERY> WHERE EXISTS(subquery)**
- Logical execution flow of EXISTS statement.
  - First, it retrieves the single row in main query.
  - Second, check for its existence in the subquery, using the column present in the subquery select statement.
  - Third, if it is present in subquery, evaluate it as True and retrieve it; Otherwise evaluate it as False and skip it.
  - <Do this for all the row, one by one>

Special Operator

➢ **ALIAS**

- ALIAS is used to **rename column** as to display it in the data output.
- ALIAS is also used to **rename Table** to refer it using alias in the other places of the querying statement.
- "AS" Keyword is used to rename the column or table. Syntax := **old_name AS new_name**.

➢ **CONCATENATION**

- **"||"** Double Pipe operator is used to Concatenate String/Text of columns
- Syntax := text_column_1 **||** text_column_2

- We can use LITERAL as ['literal_text'] also to concatenate specific text creating our own unique pattern.
- Syntax := text_column_1 || 'literal_text' || text_column_2. E.g. First_name || ' ' || Last_name

➢ **REGEX | WILDCARD**

- We can use WILDCARD operators {'**%**','**_**','**-**','**[range]**'} to create our very specific pattern matching
- It is REGEX that is Regular Expression Pattern Matching.
- IMPORTANT - To compare the STRING using Regular Expression, we have to use comparison operator tilde "~".

OPERATOR OVERVIEW

OPERATORS are used in the WHERE clause, to retrieve data which satisfies the filtering condition.

⇨ ARITHMETIC OPERATOR = Arithmetic operator are used to perform arithmetic operation on columns to form some type of expression.

⇨ COMPARISONAL OPERATOR = Comparison Operator are used to compare different values in the Where clause to specify the filtering conditions.

⇨ LOGICAL OPERATOR = Logical Operator are used to perform logical operations on the condition of WHERE Clause, to execute multiple and complex condition.

⇨ FILTERING OPERATOR = Filtering Operators are special type of operators used in the Where clause with specific use case. Like → BETWEEN = used to compare value within range of values. IN = used to check whether the value exists with any of the value from within the mention set. LIKE = Like is used to match the value with some form of regular expression. ANY | ALL = used to compare specific value with multiple or all the values with the mentioned set values. IS NULL = check whether the value is NULL or not. EXISTS = EXISTS is used to check the row existence within the subquery.

⇨ SPECIAL OPERATOR = They are special operators for columns which are not limited to just WHERE Clause. It can be applied to columns from the SELECT Clause also. ALIAS = Alias is used to rename the column or table with new temporary name to refer it with shorthand in the query or to remove the ambiguity. CONCATENATION "||" = concatenation operator is used to concatenate string/text values of the columns with or without some specific LITERALS. WILDCARD = They are used within the Regex to form some type of PATTERN to compare with value using LIKE operator mentioned above.

FUNCTIONS

- ⇨ DATETIME
  - DATETIME datatype
  - INTERVAL
  - EXTRACT
  - DATE_TRUNC
  - AGE
  - TO_CHAR
- ⇨ MATH
- ⇨ STRING
- ⇨ TYPECASTING
- ⇨ AGGREGATION
- ⇨ ARRAY_AGG
- ⇨ WINDOW
  - AGGREGATE
  - RANKING
  - EXTRA
- ⇨ NULL
  - COALESCE
  - NULLIF
- ⇨ CASE

```
/* --- 3. FUNCTIONS --- */
'PRIMARY DATATYPE' FUNCTIONS
- TIME FUNCTIONS
- MATH FUNCTIONS
- STRING FUNCTIONS
- CAST FUNCTIONS ( Implicitly "CAST" | Explicitly "TO_DTNAME")
'COLUMN' FUNCTIONS
- {"DISTINCT", "AGGREGRATE", "STRIN_AGG/ARRAY_AGG"}
'SCALAR' FUNCTION
- WINDOW FUNCTION { [AGGREGATE_FUN() | RANKING_FUN()] "OVER" (Partition) }
- NULL FUNCTIONS {Related to NULL - "COALESCE", "NULLIF"}
- CASE FUNCTION (Customized Function using LOGIC EXPRESSION)
```

DATETIME

TIMESTAMP Datatype

```
/* As per the Documentation --> Shows the value of a Run-Time Parameters. */
SHOW ALL; -- Show the Application and its settings and the description.

/* Shows the Current TIMEZONE in which we are operating. */
SHOW TIMEZONE; --Asia/Calcutta--

SELECT NOW();          --# Shows Timestamp with Time Zone w.r.t GMT.
SELECT TIMEOFDAY();    --# Same as NOW() but in more readable format as a text with Day and IST TZ. --
SELECT CURRENT_TIME;   --# Sub string of Above Function
SELECT CURRENT_DATE;   --# Sub String of Above Function
```

- Not Specific to SQL Functionality (of Data related). But gives general information about Time.
- It reports back general Time and Date Information.
- PostgreSQL have TIME, DATA, TIMESTAMP, TIMESTAMPTZ, INTERVAL datatypes for date related information.

INTERVAL

- There are "TIME INTERVAL" "ADD | SUBTRACT" function to **add or subtract time or date with Time Column**.
- PostgreSQL provides easy Addition and subtraction with dates using INTERVAL datatype. Main advantage with Postgre's INTERVAL addition or subtraction is, we don't have to explicitly mention anything while adding or subtracting time with TIMESTAMP, not even Datatype INTERVAL. Postgre understand that on its own.

Operator	Example	Result
+	date '2001-09-28' + integer '7'	date '2001-10-05'
+	date '2001-09-28' + interval '1 hour'	timestamp '2001-09-28 01:00:00'
+	date '2001-09-28' + time '03:00'	timestamp '2001-09-28 03:00:00'
+	interval '1 day' + interval '1 hour'	interval '1 day 01:00:00'
+	timestamp '2001-09-28 01:00' + interval '23 hours'	timestamp '2001-09-29 00:00:00'
+	time '01:00' + interval '3 hours'	time '04:00:00'
-	- interval '23 hours'	interval '-23:00:00'
-	date '2001-10-01' - date '2001-09-28'	integer '3' (days)
-	date '2001-10-01' - integer '7'	date '2001-09-24'
-	date '2001-09-28' - interval '1 hour'	timestamp '2001-09-27 23:00:00'
-	time '05:00' - time '03:00'	interval '02:00:00'
-	time '05:00' - interval '2 hours'	time '03:00:00'
-	timestamp '2001-09-28 23:00' - interval '23 hours'	timestamp '2001-09-28 00:00:00'
-	interval '1 day' - interval '1 hour'	interval '1 day -01:00:00'
-	timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00'	interval '1 day 15:00:00'
*	900 * interval '1 second'	interval '00:15:00'
*	21 * interval '1 day'	interval '21 days'
*	double precision '3.5' * interval '1 hour'	interval '03:30:00'
/	interval '1 hour' / double precision '1.5'	interval '00:40:00'

EXTRACT

- Allows us to **'extract' any datetime attribute from DATETIME component**. Whether it be YEAR, MONTH, HOUR, SECXONDS, etc.
- Can Extract YEAR, MONTH, DAY, WEEK, QUARTER, DOW
- It takes an datetime column and return an INTEGER value (day, month, year, hour, minute, seconds).
- Syntax :- **EXTRACT(DATE_ATTRIBUTE FROM column_name)**

```
SELECT EXTRACT(YEAR FROM payment_date) AS Payment_year FROM payment;
SELECT EXTRACT(MINUTE FROM payment_date) AS Payment_year FROM payment;
SELECT EXTRACT(YEAR FROM payment_date) AS pay_year, EXTRACT(MONTH FROM payment_date) AS pay_month FROM payment
-- Can Also make use of ALIAS function "AS" to Rename the Resultant Column of EXTRACT Function.
```

DATE_TRUNC

- It **truncates the given TIMESTAMP to its given attribute's GRANULAR level**. Meaning, if we give the attribute 'month', all the attribute after 'month' will set to their default state.
- It takes a timestamp and RETURN the timestamp.
- Syntax :- **DATE_TRUNC("DATE_ATTRIBUTE", column_name) FROM table;**

```
SELECT DATE_TRUNC('MONTH', payment_date) AS month FROM payment;    -- Will truncate till MOMNTH
SELECT DATE_TRUNC('DAY', payment_date) AS month FROM payment;      -- Will truncate till DAY
SELECT DATE_TRUNC('MINUTE', payment_date) AS month FROM payment;   -- Will truncate till MINUTES
```

AGE

- Calculate and **return the 'Current Age'** from the given Timestamp.
- It includes days as well as time from the Timestamp till present time.
- Syntax :- **AGE(time_col)**

```
SELECT AGE(payment_date) FROM payment;
```

TO_CHAR

- It is the general function to convert any data type to character/string or text datatype. But it's very powerful with timestamp to string conversion.
- Here we are using it for converting TIMESTAMP DataType to CHARACTER DataType.
- Useful for TIME formatting, as PostgreSQL provide many different PATTERN FORMATTING to convert it into. We can play with it and convert it to any Different Patterns as String code as listed out in the Documentation.
- It can also make use of ALIAS function "AS" to rename the resultant Column.
- Syntax :- SELECT TO_CHAR(date_column, 'String_code_pattern') FROM table;

```
SELECT TO_CHAR(payment_date, 'MONTH  YYYY') AS month_year FROM payment;
SELECT TO_CHAR(payment_date, 'mon/dd/YY') FROM payment;
SELECT TO_CHAR(payment_date, 'dd/mm/YY') AS indian_date FROM payment;
```

There are various number of Formats. So make sure to check out the documentation.

MATH

- There are many and almost all the mathematical operators or function available to us for being applied on numeric related Columns.
- Check out the **Documentation Section 9.3** for all the Mathematical Function. **Simple, Random, Trigonometric, advanced (log, Round, Abs, ...)**
- Just apply Mathematical formula or function on columns in SELECT statement.
- Syntax :- **SELECT Math_fun(columns/columns-expression) FROM table;**
- We can apply function on the where clause also.

- List of few Mathematical Functions
  SQRT(), PI(), SQUARE(), ROUND(), EXP(), LN/LOG2/LOG10(), GREATEST/LEAST(multiple columns), POWER(Column, n), CEILING()/FLOOR(), ABS(), Trigonometric()

```
SELECT ROUND(rental_rate/replacement_cost*100,2) AS rental_pct FROM film;
```

STRING

- PostgreSQL provide many String function and operators to Edit, Combine, and Alter Text Data. Refer **Documentation Section 9.4.**
- Refer **Documentation Section 9.7** for **Regular Expression** and Pattern Matching.
- Syntax :- **SELECT String_fun(columns) FROM table;**
- We can apply function on the where clause also.

- Some common functions

LOWER(), UPPER(), CONCAT(), LENGTH(), LEFT(), RIGHT(), STRCMP(), TRIM(), Paddings(), LPAD()
SUBSTR(str, starting_pos, end_pos), REPLACE(string, old_substring, new_substring)

```
--# String Functions
SELECT LENGTH(first_name) FROM customer;

--# Concatenation operator := "||".
SELECT first_name || ' ' || last_name AS full_name FROM customer;

--# Concatenation Operator, LITERALS and String Functions --> Making up our own string patterns AS new_column.
SELECT upper(first_name) || ' ' || upper(last_name) AS full_name FROM customer;
SELECT left(first_name,1) || '.' || last_name || '@pearls.com' AS employee_email FROM customer;
```

CASTING | TYPECAST

- CAST operator let us **convert from one datatype to another**.
- Keep in mind, it must be reasonable to cast the datatype. Meaning it should be **compatible**. i.e. not every instance of data type can be casted. E.g. Text 'five' cannot be casted to INTEGER 5. But Text '5' can be casted.
- We can only cast whole column datatype in a table instead of specific instances of table.
- Syntax := **CAST(col_name AS new_data_type)**
- **PostgreSQL specialized CAST Operator** := SELECT YEAR::INTEGER. --# **"::"** double colon here means "Casted as".

```
SELECT CAST('5' AS INTEGER);
SELECT '5':: INTEGER;

--# Suppose here we want to find the length of inventory_id. We cannot do so for INTEGER Datatype.
--# So we cast it and then perform the String function for finding length.
SELECT CHAR_LENGTH(CAST(inventory_id AS VARCHAR)) FROM rental;
SELECT CHAR_LENGTH(inventory_id::VARCHAR) FROM rental;
```

COLUMN FUNCTION

COLUMN FUNCTION are the function type, in terms that is not a scalar function, as it did not take an individual Row value as an input and execute function to return an output. Instead, it take multiple value of specific Column as an Input and return some output for column itself.

DISTINCT already mentioned above (SELECT Extension)

AGGREGATION FUNCTION

- Main idea behind an Aggregation Function is to take all the Inputs from the column and provide a Single value, known as **'MEASURE'**. Mostly an output from a **mathematical function**. Or a **mathematical measure**
- Aggregation Function to be applied, is placed in SELECT CLAUSE as Column to be Retrieved.
- **SELECT AGG_FUN(col_name) FROM table_name;**
- AGGREGATE FUNCTION LIST - MAX(), MIN(), AVG(), SUM(), COUNT(), FIRST(), LAST()

```
SELECT MAX(replacement_cost), MIN(replacement_cost), COUNT(replacement_cost),
    ROUND(AVG(replacement_cost),2) FROM film;
```

ARRAY_AGG

- **STRING_AGG** = String Aggregate Function concatenates the String from the given column in a single list, separated by comma
- It is important to remember here, that many a times it will give error as it is built on string datatype. So make sure to CAST it always as per required datatype. This is general problem for many function. So every time there is a hint of mismatch argument type, try executing after CASTing.

- **ARRAY_AGG** = Array Aggregate Function combined all the column values and return it as a single Array.

- It does not depend on datatype, as it is storing field values as an element in an array, thus all datatypes are supported.

```
--# Retrieve all the Genres|Category Type within Single Cell.
SELECT STRING_AGG (name::varchar, ', ' ORDER BY name) Genres FROM category;
SELECT ARRAY_AGG (DISTINCT name::varchar) Genres FROM category ;

--# Fetch all the movie names (in a single cell) in which the particular actor has worked in
SELECT first_name || ' ' || last_name actor_name, ARRAY_AGG(title::varchar) movies
FROM film f INNER JOIN film_actor fa ON f.film_id = fa.film_id
INNER JOIN actor a ON a.actor_id = fa.actor_id
GROUP BY actor_name

--# Fetch all the Actors which had appeared in particular Film.
SELECT title movies, ARRAY_AGG(first_name || ' ' || last_name) actors
FROM film f INNER JOIN film_actor fa ON f.film_id = fa.film_id
INNER JOIN actor a ON a.actor_id = fa.actor_id
GROUP BY movies
```

WINDOW FUNCTION

- **WINDOW FUNCTION** creates **PARTITIONS "OVER" the set of records**, separated by Category or classes of the given Column, to perform either AGGREGATION FUNCTION or RANKING FUNCTION. It is just like GROUP BY in a sense, but execution and output is somewhat different.
- WINDOW FUNCTION partitions per category is exactly same as GROUP BY, when it creates PARTITIONS before performing AGGREGATION FUNCTION.
- First difference of WINDOW FUNCTION with GROUP BY is in terms of output. It creates a new column and LOG the result of aggregation on each of the rows or records (for the respective partition).
- This is unlike GROUP BY, as it merge these records and just logs and display the Aggregated result with the respective category or class. i.e., if you remember that with GROUP BY we cannot SELECT column which are not in GROUP BY. But this is not the case with WINDOWS
- We have to select the type of function we are going to execute with WINDOW function or on a window.
- We use **OVER()** for WINDOW FUNCTION.

- Syntax: = **RANKING/AGGREGATION_FUN**(col_name_2) **OVER** (**PARTITION BY** category_column_name **ORDER BY** col_name_3)
  IMPORTANT POINTS TO REMEMBER ==>
  - **'category_column_name'** is the column by which we want to create partitions. Basically, it's a grouping column.
  - **'col_name_2'** is the column on which we want to apply WINDOW function (AGGREGATION).
  - **'col_name_3'** is the column by which we want to SORT on 'in these newly created partitions'.
- ORDER BY, i.e. Sorting is compulsory if we are applying RANKING FUNCTION, as logical.
- If we don't mention 'PARTITION BY category_col', WINDOW FUNCTION will consider whole table as one partition
- Keep in mind, **it generates new column for the resultant outputs.**

ORDER BY special execution flow

- **ORDER BY** changes the **underlying way or logic or execution flow** by which the aggregation function works on that column. READ the following paragraph carefully.
  We should be careful when applying AGGREGATION FUNCTION as WINDOW "With ORDER BY". As Underlying Execution is somewhat different. **First**, It creates the partition; **Second**, sort on sorting column; **Lastly**, THEN **apply AGG_FUN row by row, from first row, till the pointer row**, and logs the result to that row.
  i.e., Aggregated result is not same for all the rows in a partition.

Aggregated Result for the pointer row is the Aggregation FROM 'the starting row' TO 'the pointer row' in the sorted row order. IN A SENSE, it is **kind of ROLLING function**.
E.g. Suppose value in the Salary column are in the order as - 50, 70, 120, 80, 200, ...
then the aggregated result (avg) column order will be - 50, 60, 80, 80, 104, ...

Function that we can use with Window or OVER() (apart from aggregation function)

- RANK(), DENSE_RANK(), ROW_NUMBER(), PERCENT_RANK(), CUME_DIST()
- LAG(), LEAD()
- FIRST_VALUE(), LAST_VALUE(), NTH_VALUE(n)
- NTILE(n) --> Classifies the Sets of records into n buckets

Rolling Window of n Periods

Window function not only provide WINDOW for whole of the table TILL current row. BUT it also provides WINDOW of specific number of rows from current ROW, i.e. rolling window with n periods. Syntax: =

- OVER(PARTITION BNY column(s) ORDER BY column(s) **ROWS BETWEEN n PRECEDING AND CURRENT ROW**)
- OVER(PARTITION BNY column(s) ORDER BY column(s) **RANGE BETWEEN interval 'n days' PRECEDING AND CURRENT ROW**)

```
-------------------------------------
-- AGGREGATING FUNCTION with WINDOW --
-------------------------------------

--# With the Help of GROUP BY
SELECT rating,ROUND(AVG(replacement_cost),2) from film
GROUP BY rating
ORDER BY AVG(replacement_cost);

--# With the Help of OVER (PARTITION BY)
SELECT title,rating, AVG(replacement_cost) OVER (PARTITION BY rating) AS avg_replacement_cost
FROM film ORDER BY avg_replacement_cost;
--# "MAKE NOTE", How the Aggregated result is applied to all the records, instead of just combined category.
--# also we can SELECT title which is not possible with the GROUP BY call.


SELECT title, length, rating, AVG(replacement_cost) OVER (PARTITION BY rating ORDER BY length ) AS avg_replacement_cost
FROM film ORDER BY rating, length;
--# "MAKE NOTE", how the aggregated result value of AVG(replacement_cost) changes when we ORDER BY length
--# even though, they belong to same partition of Rating Category.
```

RANKING FUNCTION with WINDOW

- It 'Ranks' or assign Serial Number to RECORDS/ROWS.
- ORDER BY or SORTING is necessary for Ranking Functions. Otherwise it won't make sense for RANKING.
- We'll look at three RANKING FUNCTION - RANK(), DENSE_RANK(), ROW_NUMBER()

- /* -------- RANK() -------- */
- It Assign Ranks to Rows. In case of **ties**, it assigns same rank. But when going to the next rank, it **skips** those many RANKS with same ranks. E.g. 1,1,1,4,4,6, ...

- /* ----- DENSE_RANK() ----- */
- It Assign Ranks to Rows. In case of **ties**, it assigns same rank. But when going to next rank, it **doesn't skip** any RANKS with same ranks. E.g. 1,1,1,2,2,3, …

- /* ----- ROW_NUMBER() ----- */
- It Assign Rank to Rows in **SERIAL NUMBER**. It Doesn't take account of TIES in this. E.g. 1,2,3,4,5,6, ...

```
SELECT first_name || ' ' || last_name full_name, ROUND(SUM(amount),0) total_payment,
RANK() OVER (ORDER BY ROUND(SUM(amount),0) DESC) AS rank,
DENSE_RANK() OVER (ORDER BY ROUND(SUM(amount),0) DESC) AS dense_rank,
ROW_NUMBER() OVER (ORDER BY ROUND(SUM(amount),0) DESC) AS row_number
FROM customer JOIN payment ON customer.customer_id = payment.customer_id
GROUP BY full_name
```

NULL FUNCTION

When performing arithmetic or mathematical functions on **NULL**, it results in '**error'** or 'NULL' answer. So it's good practice to use these NULL functions to solve around these NULLs or **replace these NULLs** using some technique, and continue our execution process. We'll look at two NULL functions, namely COALESCE(), NULLIF()

➢ COALESCE

- COALESCE function accepts an unlimited number of arguments. It returns the **first argument among the argument list WHICH IS "NOT NULL**". E.g. COALESCE (1,2,3) ==> 1, COALESCE (NULL, a*b, 10) ==> 10 if a*b is NULL.
- What this basically means is that it accepts an input, check if it is an NULL, if it is then replaces it with next argument, if that is also NULL, goes to next argument until NOT NULL value is found and then return it.
- If all the argument in the coalesce is NULL, its output will also be NULL.
- Now you are most probably wondering how it is used in Query call. It is the most common useful function when querying a table using mathematical operator or function and it is consisting of NULL Values. As NULL datatype values is not compatible with INTEGER DATATYPE. Those is replaced with next argument with the coalesce function. Thus our mathematical function/operator keeps on working and produced desired result for all record.
- Syntax := **COALESCE(column_name,arg_1,arg_2,...).** What it will do is that for values of all the Rows/Record for that specified column is IF NULL then replace it with next argument without altering or affecting the original Table.
- E.g. (col_1 - COALESCE(col_2,0)) ==> **(Price - COALESCE(Discount,0))**. So while performing this mathematical operation, if NULL is encountered in Discount Column, then in spite of giving result as NULL, it will go to next argument, which is 0 here, and continue to perform the math operation efficiently.

➢ NULLIF

- NULL IF takes 2 value as an argument and **returns NULL if both are equal otherwise returns first argument** as output.
- Syntax := **NULLIF (arg_1, arg_2)**
  - **NULL** --> If arg_1 = arg_2 return NULL;
  - **arg_1** --> If arg_1 != arg_2 return arg_1;
- This became useful when NULL value would cause an error or undesired results.
- Example scenario, we can compare the column values with specific values to output NULL, instead of generating an error. NULLIF(col_name, 0), if we are using the specified column in denominator for arithmetic expression, then this keyword will save us from giving 'divide by 0' error.

```
SELECT
SUM(CASE WHEN department = 'A' THEN 1 ELSE 0 END)/
NULLIF(SUM(CASE WHEN department = 'B' THEN 1 ELSE 0 END),0)
AS department_ratio
FROM depts;
```

CASE FUNCTION

- CASE, in my words, is "**Customized Scalar Function**" based on logical expression.
- CASE is **analogous to IF/ELSE** or logical statements from any other programming language. That is, it generates result based on certain conditions

- There are two ways to use CASE - "**general CASE**" (analogous to IF/ELSE) or "**CASE expression**" (analogous to SWITCH). Both leads to same result.
- CASE is placed inside a SELECT phrase almost as it's an another column. Thus it outputs result in another separate column.
- "General CASE" is same as IF/ELSE. It is more flexible and most used as any type of expression can be evaluated. Syntax :=

```
SELECT
    CASE
        WHEN condition_1 THEN result_1
        WHEN condition_2 THEN result_2
        ELSE some_other_result
    END
FROM table_name;
```

- "CASE Expression" syntax is same as SWITCH/CASE. It first evaluates an expression then compares the result with each value in the WHEN clause Sequentially. It is not flexible and usually used for categorical columns. As it cannot evaluate any type of expression. Syntax :=

```
SELECT
    CASE expression
        WHEN value_1 THEN result_1
        WHEN value_2 THEN result_2
        ELSE some_other_value
    END
FROM;
```

- Keep in mind, that **CASE results the output in newly created separate column.**

```sql
SELECT customer_id,
CASE
    WHEN (customer_id <= 100) THEN 'Platinum'
    WHEN (customer_id BETWEEN 101 AND 200) THEN 'Gold'
    WHEN (customer_id BETWEEN 201 AND 300) THEN 'Silver'
    ELSE 'Bronze'
END
AS customer_type
FROM customer;


SELECT customer_id,
CASE customer_id
    WHEN 1 THEN 'Gold Medal'
    WHEN 2 THEN 'Silver Medal'
    WHEN 3 THEN 'Bronze Medal'
    WHEN 4 THEN 'Certificate'
    WHEN 5 THEN 'Certificate'
    ELSE 'Participants'
END
AS Reward
FROM customer WHERE customer_id < 10;
```

```sql
--# Use cases for this. Note here, though this can be done using another Operators or Keywords,
--# but this gives us more tool or variation to play with. And also we can play with formatting here.
SELECT
SUM (CASE rental_rate
        WHEN 0.99 THEN 1
        ELSE 0
END) AS bargained_payments,
SUM (CASE rental_rate
        WHEN 2.99 THEN 1
        ELSE 0
END) AS regular_payments,
SUM (CASE rental_rate
        WHEN 4.99 THEN 1
        ELSE 0
END) AS overcharged_payments
FROM film;
```

FUNCTION OVERVIEW

FUNCTIONS are applied on the Columns for manipulation and analysis purpose. And in the data terms, for feature engineering. They are of Scalar or Column Type. They are applied on the Columns, mostly in the SELECT Clause or WHERE clause.

- ⇨ TIME FUNCTIONS = Time functions are used to perform time operation on time datatype columns. Like to extract specific field of Time value or to find the age or interval from some time to another time, or to track record of when the particular transaction happens.
- ⇨ MATH FUCTIONS = Maths function is used to perform Math operation on columns for analysis purpose and feature engineering.
- ⇨ STRING FUNCTION = String Function are used to perform string operations on text columns. Like to convert or extract specific type of pattern from text, or to concatenate texts, so on.
- ⇨ TYPECAST FUNCTIONS = CAST Function allows us to convert the datatype of Column from one type to another type for various analysis and manipulation purpose.
- ⇨ COLUMN FUNCTIONS = Theses functions are special type of function, in terms that they take multiple values of column as an Input and result in some output (either single with Aggregation or multiple with DISTINCT or Ranking.) AGGREGATION FUNCTION = These executes aggregation operation such as SUM, MAX, MIN, COUNT, AVG on multiple values of column per category to give aggregated result.
- ⇨ WINDOW FUNCTION = These functions are executed over the partition set of records. In a sense, they perform aggregated or Ranking function over the set of records per PARTITIONS and log the result to each record in a new column. RANKING function deals with various ranking types functionality.
- ⇨ NULL FUNCTIONS = These functions are used to deal with the NULL Values in the column or with the possible error result while performing arithmetic or maths operations.
- ⇨ CASE FUNCTION = It is the customized function that we can create of our own, based on logical statements and conditions. Just like IF/ELSE, but on every record value of column to transform it. Its output is stored in new column. Example - dividing continuous numerical values into distinct categories or to convert categories to some higher level of hierarchy, or to convert values into binary classification, and many more.

JOIN

JOIN Theoretical Concepts

JOIN TYPE Queries


```
/* --- 4. JOINS --- */
- "JOIN OVERVIEW" ('JOIN THEORY', 'JOIN TYPES', 'JOIN FORMS', "ON", "USING")
- INNER JOIN
- OUTER JOIN
- OUTER JOIN except INTERSENT | INNER
- LEFT JOIN
- LEFT JOIN except INTERSENT | INNER
- RIGHT JOIN
- RIGHT JOIN except INTERSENT | INNER
- SELF JOIN
- 'SET OPERATIONS' {"UNION", "INTERSECT", "EXCEPT"/"MINUS"}
```

JOIN Theoretical Concepts

JOIN THEORY

==> **What is JOIN and Why it is Used.**

- JOIN allows us to **Combine** or **Join multiple tables together**, as individual table consists of limited sets of information.
- JOIN is used to execute 'Table Manipulation' by combining different Datasets or Tables.
- With JOIN, we can **access and analyse data** which are **stored across different tables** in a database.
- JOIN **connects columns (attributes) from one table to another tables** using some relation or mapping.
- What is **Mapping**. We need some **ways to connect information from one table to another table's information**. It does not make sense to connect every data from one table to every data in another table. "**Primary Key-Foreign key**" relation allow us to do exactly that.
- **Primary key** is the unique identifier for a particular Table. **Foreign Key** are the Primary key from another table. Thus, Foreign key notifies the row, with which unique data it has connections to in the other table.
- You'll find that this **Primary Key and Foreign key** are the "**MATCHING COLUMN**" between 2 tables.

==> **General JOIN SYNTAX**

FROM left_Table JOIN_TPYE right_Table ON Conditon/Expression; --# 2 Table Join

FROM left_Table JOIN_TYPE right_Table ON Conditon/Expression JOIN another_right_Table ON Condition/Expression; --# 3 Table Join

Analogy

left_Table is also referred as **Table A** or **Table 1**.
right_table is also referred as **Table B** or **Table 2**.

==> **ON Phrase** ==> Consideration for Joining Table.

- Now, there are two main things to CONSIDER for Joining Two Tables.
  First, we join Two Tables based on the "Matching Column (Same Attribute with Same Datatype)"
  Second, we join Two tables with some "Condition | Expression" on "this Matching Column".
  (Though Mostly the Condition is Equality --> EQUI | NORMAL JOIN)
  (It can be Expression with any Condition and Logical Operator also --> CONDITIONAL JOIN)
- This is specified by "ON Phrase of JOIN".
- This Matching column is nothing but the Primary Key-Foreign Key relation.

==> **INFO SETS in JOINED TABLE** (Columns Available in Joined Table)

- As there are two tables, there are 2 Sets of Information (Column or Attribute List) Available to us in the Joined Table.
  [A | Left] INFO SET ==> From the 'left_Table'
  [B | Right] INFO SET ==> From the 'right_Table'
- In Essence, there are three type of columns present in the **"SELECT *"** FROM Joined Table.
  - "Matching Column"
  - "Columns from Information Set A" or "**A.cols**"
  - "Columns from Information Set B" or "**B.cols**"

==> **ROW COMPONENTS in JOINED TABLE**

- After these consideration, we have to look at **ROW COMPONENTS** Present in the Joined Table.
  1st Row Constructs (Component 1) - Those Records which are "**Unique**" to **left_Table** ON the Matching Column.
  2nd Row Constructs (Component 2) - Those Records which are "**Common**" to **Both the Tables** ON the Matching Column.
  3rd Row Constructs (Component 3) - Those Records which are "**Unique**" to **right_Table** ON the Matching Column.
  IMPORTANT →
  What do we mean by '**unique'** to Left Table or Right Table → While Joining Tables ON matching column, it may happen that some field value which are present in the Left Table 'does not exist' at all in the Right Table <and Vice Versa>. That means there is no mapping available to us for those records in the Right Table <and vice versa>. For those records, the information set from the right table will be set as NULL <and vice versa>.
  ***(We are defining UNIQUE or COMMON on the basis of Matching Column only, as this column is the one in the left and right table, that we are using for MAPPING or connection of two tables.) ***

- Based on above Derived Components, we can make some **LOGICAL STATEMENT** for **INFORMATION SETS** available to us in the Joined Table.
- **Logically**,
  - **Component 1** will only consist of records which are not present in the Right Table, i.e., "**Records from the Left_Table only**" ==> so after joining, Component 1 will only have the "**Left_Table Columns**" with "**Right_Table Columns set as NULL**".

- **Component 3** will only consist of records which are not present in the Left Table, i.e., "**Records from the Right_Table only**" ==> so after Joining, Component 3 will only have the "**Right_Table Columns**" with "**Left_Table Columns set as NULL**".
- **Component 2** will consist of records which are present in both the table, i.e., "**Records common to BOTH the Table**" ==> so after Joining, Component 3 will have "**BOTH the Left and Right Table Columns available**" with "**NONE AS NULL**".

***(Again, relations like 'unique' and 'common' are on the basis of 'Matching column' from the left and right table) ***

==> **Programming equivalent LOGIC CONDITION (based on above Logical Statement)**

- Using the same Logic above (on basis of Column Sets Available) we can create its Programming Equivalent Logic Condition (within WHERE CLASUE) to derive any bucket type of Row Components. Let's see how:
  Row Component 1's Logic Condition → WHERE "b.info IS NULL" (# Table B columns IS NULL)
  Row Component 3's Logic Condition → WHERE "a.info IS NULL" (# Table A columns IS NULL)
  Row Component 2's Logic Condition → WHERE "a.info IS NOT NULL AND b.info IS NOT NULL" (Both Table A cols and Table B cols IS NOT NULL)
- We can make any combination of Row Components using these Programming Logic Conditions.
- We'll see how it is useful later in this section.

JOIN TYPES

==> **JOIN TYPES**

- **INNER JOIN** ==> (Records present in both the Table with both Column Info Sets Available). i.e., **(Component = 2)**
- **OUTER JOIN** ==> (Unique Records from the Left Table + Unique Records from the Right Table + Intersection Record from both the Table). i.e., **(Component = 1+3+2)**
- **OUTER JOIN without INNER** ==> (Unique Records from the Left Table + Unique Records from the Right Table). i.e., **(Component = 1+3)**
- **LEFT JOIN** ==> (Unique Records from the Left Table + Intersection Record from both the Table). i.e., **(Component = 1+2)**
- **LEFT JOIN without INNER** ==> (Unique Records from the Left Table). i.e., **(Component = 1)**
- **RIGHT JOIN** ==> (Unique Records from the Right Table + Intersection Record from both the Table). i.e., **(Component = 3+2)**
- **RIGHT JOIN without INNER** ==> (Unique Records from the Right Table). i.e., **(Component = 3)**
  ***(Again very Important Remembrance Point - All the Set Relation of Records are ON BASIS OF MATCHING COLUMN ONLY) ***

==> JOIN TYPES Derivation

- From above understanding of JOINS, we can categorize Join into mainly **Two Types** and all the rest can be derived from it.
  First, **Records COMMON to both the Table**.
  Second, **Records UNIQUE to either of the table**.
- If we are able to understand these 2 Components, we can understand any of the JOIN types; and can derive as per the situation.
  - **"INNER JOIN (2)"** <Records COMMON to both the table>
  - **"OUTER JOIN without INNER (1+3)"** <Records UNIQUE to either of the Table>

- **Derivation of All Join from above 2 Joins**

- INNER JOIN = INNER JOIN itself (2)
- OUTER JOIN = OUTER with INNER (1+2+3)
- LEFT JOIN = Left Part of OUTER with INNER (1+2)
- RIGHT JOIN = Right Part of OUTER with INNER (3+2)
- LEFT JOIN without INNER = Left Part of OUTER (1)
- RIGHT JOIN without INNER = Right Part of OUTER (2)

- **Derivation of all Join through Query**
  Now to derive all joins through Query, use the LOGICAL CONDITION in WHERE CLAUSE.
  - INNER = INNER
  - OUTER = OUTER
  - LEFT = LEFT
  - RIGHT = RIGHT
  - OUTER except INNER = OUTER WHERE b.info is NULL AND a.info is NULL
  - LEFT except INNER = LEFT WHERE b.info is NULL (OR WHERE a.info IS NOT NULL)
  - RIGHT except INNER = RIGHT WHERE a.info is NULL (OR WHERE b.info IS NOT NULL)

- ***** It's to understand whole JOIN Concept just with "3 ROW COMPONENTS and 2 COLUMN SETS" *****.

JOIN FORMS

- **NORMAL JOIN** -
  - When there is only one **single matching column** in both the Joining Tables, then the JOIN of two tables with "**EQUALITY CONDITION**" is **Normal Join**.
  - In this Join type, the Two Tables can be joined without even mentioning Matching Condition by naming join type as "NATURAL JOIN", as there is only one matching column and condition is by default Equality. It should be avoided at all cost, when there is more than one matching column, as it creates an unwanted result.
- **EQUI JOIN**
  - When there is **more than one matching column** in two tables, then the joining of two table on **'Equality Condition**' is called "**EQUI JOIN**".
  - EQUI JOIN is the very specific case of Normal Join.
  - In this type of Join, two tables can be joined by 'Explicitly' mentioning matching column.
  - Now, if the **matching column name is same** in both the table, we can use "**USING**" Keyword as **USING(matching_column_name)** to Join tables.
  - But if **matching column has different names** in different tables, then we have to use "**ON**" Keyword to write the matching expression. **ON table_A.col_name = tabl_B.col_name**
- **CONDITIONAL JOIN**
  - When we have to join '**not based on Equality Condition'** then the joining is called **Conditional join**.
  - We can use "**ON**" Keyword only with full expression, as USING use only Equality condition to join.
- **CROSS JOIN**
  - Cross Join **joins every row of one table with every row of another table**.
  - That is, it is resulting **m*n rows**. where 'm' and 'n' are the number of rows of each table.
  - It is used when we don't know the relation of two tables, and thus we join every record with every record of other table.
  - It is also known as Cartesian Product or Cartesian Join.
  - To CROSS JOIN, simply mention both "**Table name separated by Comma"**.

- Everything else except the "Conditional Matching Condition/Expression" is clear from all the above theories.
- IDEAS for COMPREHENDING the "Conditional Matching [Condition | Expression]".
  - Generally, we perform join on "Primary Key" only so as to get additional information for Particular Record. And it is logical that for primary/foreign key we have to use EQUI Join Form only.

- But when we want to compare two records on basis of some particular feature, then we have to first create pair for such comparison (=, >, <, !=) by executing Join on that feature column, then we can filter it based on our requirement or specific analysis scenario.
- This can be best understood through examples given in Self Join.

➔ Difference between **ON** and **USING**

ON

- We join the 2 Tables on some "Matching Column" or Columns which contain same type of values.
- **"ON"** allows us to define on which column we are joining, and with which matching Condition or Expression.
- Use ON When, -
  - We are joining on same column type and with "**Equality**", but both **column has different names**.
    JOIN table_A **JOIN_TYPE** table_B **ON table_A**.matching_column_name_A = **table_B**.matching_column_name_B;
  - We are Joining on same column type but condition is "**Not Equality**", but some other "Expression".
    JOIN table_A **JOIN_TYPE** table_B **ON EXPRESSION(table_A**.matching_column **comparison_operator table_B**.matching_column);
    - E.g. = **table_A**.col_id **> 4*table_B**.col_id **AND table_A**.col_name **!= table_B**.col_name

USING

- Use USING, When -
  - The matching column in both the table has **same name** and matching condition is **equality**.
    JOIN table_A **JOIN_TYPE** table_B **USING(matching_column_name);**

JOIN QUERIES

INNER JOIN

- An INNER JOIN will result in retrieving of set of records that are **COMMON on "Matching Column" in Both the tables**.
- (Thus 2nd Row Component - Those Records which are present in both the Table with both Column sets of information available - A and B). - i.e., **Row Component "2".**
- By Default, JOIN means INNER JOIN. It is the most used JOIN for querying.
- Syntax :- SELECT * **FROM** table_A **INNER JOIN** table_B **ON** Table_A.col_match **=** Table_B.col_match;
- Table order won't matter in INNER JOIN as it is **symmetrical** in nature.
- One thing to bear in mind, that the Matching Column will appear twice in resultant Joined Tables, one from table A and one from Table B, so to remove ambiguity, explicitly mention matching_column from any one of the table AS table_A.col.

```
SELECT *
FROM payment INNER JOIN customer ON payment.customer_id = customer.customer_id;

SELECT payment_id, payment.customer_id, first_name
FROM payment INNER JOIN customer ON payment.customer_id = customer.customer_id;

SELECT customer.customer_id,first_name,last_name,address_id,staff_id,rental_id,amount,payment_date
FROM customer INNER JOIN payment ON customer.customer_id = payment.customer_id;
```

OUTER JOIN

- An FULL OUTER JOIN will result in **ALL the Set of records that ARE ON "Matching Column"** in Both the tables.
- Thus 2nd Component - those records which are present in both the tables with both set of informations available,
  + 1st Component - Data Records, unique to Table A ON "Matching Column", with A set of informations available and with B set of informations as NULLs,

+ 3rd Component - Data records, unique to Table B ON "Matching Column", with B set of informations available and with A set of informations as NULLs.

- i.e., **Row Component "1+2+3"**.
- Syntax :- SELECT * **FROM** table_A **FULL OUTER JOIN** table_B **ON** Table_A.col_match = Table_B.col_match;
- Order Doesn't matter for Full Outer Join as it is also **symmetry** in nature.

```
SELECT customer.customer_id,payment.customer_id,first_name,last_name,address_id,staff_id,rental_id,amount,payment_date
FROM customer FULL OUTER JOIN payment ON customer.customer_id = payment.customer_id;
```

➢ OUTER without INNER JOIN

- It retrieve the data Records which are **unique to each of the table ON "Matching Column**". i.e., **Row Component "1+3".**
- It is Mutually Exclusive Area of INNER JOIN.
- Syntax :- SELECT * **FROM** table_A **FULL OUTER JOIN** table_B **ON** Table_A.col_match = Table_B.col_match **WHERE Table_B.col is NULL OR Table_A.col is NULL**;
- 'Table_B.info is NULL' retrieves all the Records unique to Table A. Similarly, Table_A is NULL retrieves all the Records unique to Table B. As records unique to Table A contains B set of information as NULL and vice versa. (Explained above in Logical Statements).
- This is Complement of INNER JOIN as per the VENN Diagram. That is unique only to Table A or Table B.
- It is also symmetrical in nature.
- Scenario E.g.- With 'customer.customer_id is NULL', I'm retrieving only those components which had made transaction with the store but are not present in the customer Table in the Database. Similarly, with 'payment.customer_id is NULL', I'm retrieving those components which had not made any transaction with the store but are present in the Customer Table in the Database.

```
SELECT customer.customer_id,payment.customer_id,first_name,last_name,address_id,staff_id,rental_id,amount,payment_date
FROM customer FULL OUTER JOIN payment ON customer.customer_id = payment.customer_id
WHERE customer.customer_id is NULL OR payment.customer_id is NULL;
```

LEFT JOIN

- A LEFT OUTER JOIN will result in set of records that **ARE ON "Matching Column" in Both the tables** AND **unique to LEFT Table/Table A**.
- Thus 2nd Component - those records which are present in both the tables with both set of informations available,
  as well as, 1st Component - Data Records, unique to Table A ON "Matching Column", with A set of informations available and with B set of informations as NULLS,
  but, excluding 3rd Component - that is unique to Table B ON "Matching Column".
- i.e., **Row Component "1+2".**
- Syntax :- SELECT * **FROM** table_A **LEFT OUTER JOIN** table_B **ON** Table_A.col_match = Table_B.col_match;
- Order of tables Does Matter in LEFT JOIN as it is **not Symmetry** in nature.
- It is Symmetry to RIGHT JOIN mirrored around the INNER JOIN. That is to say, On Applying RIGHT JOIN with Reversing the order, it will result in same DATA OUTPUT.
- Example Scenario – Retrieve data for all the Customer and their payments. It doesn't matter whether they have made payment or not.

```
SELECT *
FROM customer LEFT OUTER JOIN payment ON customer.customer_id = payment.customer_id;
```

➢ LEFT without INNER JOIN

- To Retrieve Records/Entries **unique to Table A ON "Matching Column**".
- i.e., Row Component "1".
- Syntax :- SELECT * **FROM** table_A **LEFT OUTER JOIN** table_B **ON** Table_A.col_match = Table_B.col_match **WHERE Table_B.col is NULL**;

- Condition 'Table_B.col is NULL' discard those records which are present in both the tables.
- Example Scenario. – We want to find the Customer who have made the Payments. That is to say that they never transacted with us yet they are in our customer dataset. So we may would like to remove these customers.

```
SELECT customer.customer_id,payment.customer_id,first_name,last_name,address_id,staff_id,rental_id,amount,payment_date
FROM customer LEFT JOIN payment ON customer.customer_id = payment.customer_id
WHERE  payment.customer_id is NULL;
```

RIGHT JOIN

- RIGHT OUTER JOIN will result in set of records that **ARE ON "Matching Column" in Both the tables** AND **unique to RIGHT Table/Table B**.
- Thus 2nd Component - those records which are present in both the tables with both set of informations available,
  as well as 3rd Component - Data Records, unique to Table B ON "Matching Column", with B set of informations available and with A set of informations as NULLs,
  but excluding 1st Component - that is unique to Table A ON "Matching Column".
- i.e., Row Component "3+2".
- Syntax :- SELECT * **FROM** table_A **RIGHT OUTER JOIN** table_B **ON** Table_A.col_match = Table_B.col_match;
- Order of tables Does Matter in RIGHT JOIN also as it is **not Symmetry** in nature.
- It is Symmetry to LEFT JOIN mirrored around the INNER JOIN. That is to say, on Applying LEFT JOIN with Reversing the order, it will result in same DATA OUTPUT.
- Example Scenario – Retrieve all the payment details, whether or not their customer exists in the database.

```
SELECT *
FROM customer RIGHT OUTER JOIN payment ON customer.customer_id = payment.customer_id;
```

➢ RIGHT JOIN without INNER

- To Retrieve Records/Entries **unique to Table B ON "Matching Column**".
- i.e., Row Component "3".
- Syntax :- SELECT * **FROM** table_A **LEFT OUTER JOIN** table_B **ON** Table_A.col_match = Table_B.col_match **WHERE Table_A.col is NULL**;
- Condition 'Table_A.col is NULL' discard those records which are present in both the tables.
- Example Scenario. - Suppose we have to find customer who had made transactions with us in the past and are present in the payment dataset,
  but not included or incorporated in the customer Dataset. So we may like to add these customer's details in our Customer dataset.

```
SELECT customer.customer_id,payment.customer_id,first_name,last_name,address_id,staff_id,rental_id,amount,payment_date
FROM customer FULL OUTER JOIN payment ON customer.customer_id = payment.customer_id
WHERE  customer.customer_id is NULL;
```

SELF JOIN

- Self-Join is a type of query in which **'Table is Joined to Itself**' that is joined **to its copy**.
- It is useful for comparing values in a column within the same table or referring values to the same Table.
- Essentially, it is used when there is a self-relation (relation from table to itself in a Database Relational Diagram). That is when self-join is used.
- There is **no special keyword for self-join**. It is just a JOIN with itself.
- Any type of JOIN with itself will results in same output, as all the components of Venn diagram coincides.
- Also, when using Self-Join, it is **necessary to use ALIAS** for table **to remove the ambiguity** when using multiple times.
- Syntax :- **SELECT A.col, B.col FROM table AS A JOIN table AS B ON A.some_col = B.other_col;**
-

- **Matching Criteria or Condition after ON can be any type of condition** (Just like WHERE Conditions, but for JOIN, the tables column are present on either side of condition), not necessarily "Equals to"
- Example Scenario - Suppose if a table contain employee_id and report_id as to whom they report to, signifying referencing table relation to itself
  (A.emp_col (=) B.report_col) Relation is "employee to another employee" in a relational diagram.
  So to retrieving and analysing information relating to this we need to self-join so that report_id relates to emp_id.

```
/* --- Example for Problem - Retrieve the pair of films that have the same length. --- */
/* Solution Approach ==>
-- To perform this we have to join film with film itself, meaning self join film to relate one record's length to another records's length in same table.
-- To take idea of approach look at this query, we get 5 films with same length of 117:=  SELECT title, length FROM film WHERE length = 117;
-- So I have to pair them with each other as per the problem.
-- Here I can Match on length column (which is of attribute type)
-- (A.film_id != B.film_id) filters the resultant self joined table, to make sures the exclusivity of same movie pairs.
-- Also it gives us insight on how we can play with matching condition.
*/
SELECT A.title, A.length, B.length, B.title
FROM film AS A JOIN film AS B ON A.length = B.length AND A.filM_id != B.film_id;


/* --- Example for Problem - Pair of films in which 1 film have greater than 4 times the length of another film. */
SELECT A.title, A.length, B.length, B.title
FROM film AS A JOIN film AS B ON A.length > 4*B.length;

/*
IDEAS for COMPREHENDING the "Conditional Matching Condition/Expression".
- Generally, we perform join on "Primary Key" only so as to get more information for Particular Record.
  And it is logical that for primary/foreign key we have to use EQUI Join Form .
- But when we want to compare two records on basis of some particular feature,
  then we have to first create pair for such comparison(=,>,<,!=) by performing Join on feature column.
- then we can filter it based on our requirement need or specific analysis scenario.

- This can be best understood through this above examples.
*/
```

TABLE SET OPERATIONS

<Note – These are Set Relations on “Whole Records” rather than on Matching Column. There are no as such “matching column” for Table Set Operations. >

```
/* --- I've created two Table For performing Set Operations on Table --- */

--# Films whose rental rate is 0.99$.
--# Resulting in 344 records/films
SELECT title,rental_rate,length FROM film  WHERE rental_rate = 0.99

--# Films whose runtime is greater than 100 mins.
--# Resulting in 622 records/films
SELECT title,rental_rate,length FROM film WHERE length >= 100
```

➢ UNION

- It **concatenates two Table on top of each other**.
- It should be logical for this that, Selected columns should be same for both the TABLE.
- Meaning **Column Type and Order must be Compatible** for both the Table.
- Values for all records (rows) for one Table, and for the features (column) which is **not present** as per the **another TABLE**, would **result in "NULL"**.
- There is one more variation to UNION, “**UNION ALL**”
- Syntax :=

  SELECT column_names FROM Table 1
  UNION
  SELECT column_names FROM table 2;

```
--# Retrieve those films whose rental_rate is Either 0.99$ or length is greater than 100 minutes.
SELECT title,rental_rate,length FROM film WHERE rental_rate = 0.99
UNION
SELECT title,rental_rate,length FROM film WHERE length >= 100;
--# Resulting in 757 Records
```

➢ EXCEPT

- It retrieves those **records** which are **present in first query results "BUT" not exists in second query results**.
- It should be logical for this that, Selected columns should be same for both the TABLE.
- Meaning **Column Type and Order must be Compatible** for both the Table.
- MINUS operator is also same as EXCPET Operator.
- Syntax :=

```
SELECT column_names FROM Table 1
EXCEPT
SELECT column_names FROM table 2;
```

```
--# Retrieve Those films whose rental_rate = 0.99$ AND runtime is not greater than 100 minutes.
SELECT title,rental_rate,length FROM film WHERE rental_rate = 0.99
EXCEPT
SELECT title,rental_rate,length FROM film WHERE length >= 100;
--# Resulting in 135 Records
```

➢ INTERSECT

- It retrieves those **records** which are **present in first query results "AS WELL AS" exists in second query results**.
- It should be logical for this that, Selected columns should be same for both the TABLE.
- Meaning **Column Type and Order must be Compatible** for both the Table.
- Syntax :=

```
SELECT column_names FROM Table 1
INTERSECT
SELECT column_names FROM table 2;
```

```
--# Retrieve Those films whose rental_rate = 0.99$ AND runtime is greater than 100 minutes.
SELECT title,rental_rate,length FROM film WHERE rental_rate = 0.99
INTERSECT
SELECT title,rental_rate,length FROM film WHERE length >= 100;
--# Resulting in 206 Records
```

JOINS OVERVIEW

Joins are used within the FROM clause. Join are used to combine different table when we want to access informations that is stored across different tables. It is used to connect column information based on some common column/attribute (generally primary Key-Foreign Key relation). Joined Table consists of three row components or we can say row buckets. Different combinations of these 3 row components forms the basis of all the types of join. There are four forms of Joins – EQUI JOIN, NORMAL JOIN, CONDITIONAL JOIN, CROSS JOIN. ON Clause (or USING Keyword) specifies the matching condition for Joining Table. There are mainly four types of JOINS.INNER JOIN, OUTER JOIN, LEFT JOIN, RIGHT JOIN. OUTER, LEFT, RIGHT Join has a variation as “Without INTERSECT or the INNER Part”.

- ⇨ INNER JOIN = used to retrieve the records which are present in both the table in the matching column values.
- ⇨ OUTER JOIN = used to retrieve all the records, i.e. which are present in both tables or either in the left table or right table, in the MATCHING COLUMN.
- ⇨ LEFT JOIN = used to retrieve the records which are unique to Left Table as well as record common to both.
- ⇨ RIGHT JOIN = used to retrieve the records which are unique to right table as well as common to both the table.
- ⇨ SELF JOIN = use to join table with itself or its copy, to perform the comparison of record with other records from the same table. SELF JOIN is different from other type of joins as other JOINS are joined to connect pieces of information, where in self-join records are join to compare one record with other records of the same table.
- ⇨ TABLE SET OPERATIONS = all the above Join Operations are set operations onto the matching column, but these are set operations onto the whole row. And by this definition, it is logical that retrieved columns from each

table must be compatible with each other to perform set operations. UNION = used to retrieve all the records from both the table. EXCEPT = used to retrieve the set of records from one table, which are not present in another table. INTERSECT = use to retrieve records which are present in both the table.

SUBQUERY & TEMPORARY TABLE

- ⇨ **VIEW**
- ⇨ **CREATE TABLE AS**
- ⇨ **SELECT … INTO**
- ⇨ **CTEs**
- ⇨ **Recursive CTEs**

```
/* --- 5. TEMPORARY TABLE --- */
SUBQUERY (Independent & Correlated)
TEMPORARY TABLE
- VIEW
- CREATE TABLE AS
- SELECT INTO
- CTEs
- Recursive CTEs
```

SUBQUERY

SUBQUERY are of two types, Independent and Correlated. Independent Subquery don't depend on the Main query. While, Correlated Subquery have somewhat connection to main query and cannot execute on its own.

- What if we have to perform a query on the result of the another query. We may need to store query results as a variable and then can we use it, right no! But what If SQL provides a way to simplify this by directly taking the use of query results as a sub-query in the main query.
- Essentially, we are replacing any temporary variable (as in python programming language) directly with subquery.
- Parentheses are compulsory for SUBQUERY.
- Three type of output are possible with Subquery - "SINGLE VALUE" or "SINGLE COLUMN as SET VALUE" or "TABLE".
- There are two type of Subqueries. "Independent" aka "NESTED" or "CO-RELATED".
- INDEPENDENT/NESTES = If the Execution of Inner Query is INDEPENDENT of Outer Query. Inner Query Get executed first and then Outer Query.
- CO-RELATED = If the Execution of Inner/Sub Query is DEPENDENT on Outer/Main Query. Inner/Sub Query GET data(column/row) from Outer/Main Query "Row by Row", perform Execution. Then, its Results are used for Outer/Main Query to execute.
- There are three places where it can be used. - "WHERE/HAVING" Clause or "FROM" Clause or "SELECT Clause".
- Syntax :- Just replace the variable (which is nothing but output of another query) with subquery (of which it is output of) in the parentheses inside the main query.
- There is no particular SYNTAX for SUBQUERY but most common used case is ==>
  SUBQUERY within "WHERE" or "HAVING" CLAUSE -
  SELECT FROM WHERE column_name EXPRESSION_OPERATOR (SUBQUERY).
  E.g. **- WHERE salary > (Subquery_SingleValue);**
  ==> WHERE grade > 70; --# 70 is the result of subquery
  - WHERE student IN (Subquery_SetValues);
  ==> WHERE student IN ('Ram', 'Shyam', 'Ramesh', 'Suresh'); --# List is Subquery's result
  - WHERE salary > ALL|ANY (Subquery_SetValues);
  ==> WHERE salary > ALL|ANY (70000,80000,110000,150000,250000,500000);
  - WHERE EXISTS (Subquery_Table);
  ==> WHERE EXISTS T's/F's;

SUBQUERY within "FROM" CLAUSE - It is executed first for this case and is stored as TEMPORARY TABLE.

SELECT column(s) **FROM (Subquery_Table);**

SELECT column(s) **FROM (Subquery_Table)** as table_1, table_2 WHERE condition(s);

⇨ **INDEPENDENT SUBQUERY** can result in three type of Output, Namely =

1. **Table** => Used in the **FROM** Clause Only, as **'Temporary Result Table'**. Identification Syntax :=

... FROM (Subquery_Table) ...;

2. **Single Value** => Used in the **WHERE** or **HAVING** Clause only. Compulsory preceded by comparison_operator. Syntax :=

... WHERE column_value comparion_operator (Subquery_SingleValue) ...;

```
-------------------------------------------------------------------------------------------------------------
/* --- If a subquery results in a ""Single Value"" output then we can use it with comparison operator. --- */
/* Example for Problem - Retrieve Movie Titles whose Rental Rate is more than average Rental Rate. */
SELECT title, rental_rate FROM film
WHERE rental_rate > (SELECT AVG(rental_rate) FROM film);
-------------------------------------------------------------------------------------------------------------
```

3. **Single Column** (SET VALUE) => 2 Cases, As per my current knowledge, -->

First, it can be used in **'SELECT'** clause for **'printing column from another table without using JOIN'**.

... SELECT t1.c1, (SELECT t2.c2 FROM t2 WHERE matching_condition) FROM t1;

```
/*Return Film name with its corresponding Category, WITHOUT using JOIN*/
SELECT f.title,
    (SELECT
        (SELECT c.name FROM category c WHERE f_c.category_id = c.category_id)
    FROM film_category f_c WHERE f.film_id = f_c.film_id)
FROM film f;
```

Second, used in the **'WHERE'** or **'HAVING'** Clause for comparison purpose. Used with 3 Special Filtering_Operator ("IN", "ANY | ALL")

... WHERE column_value IN (subquery_Column_SetValues) ...;

... WHERE column_value comparison_operator ALL|ANY () (subquery_Column_SetValues) ...;

... WHERE EXISTS (subquery_Column_SetValues) ...;

```sql
/* ---- If a subquery results in a ""Column/LIST"" output then we can use it with 'IN',
        'ALL|ANY' or with 'EXISTS' function to check its existence. ---- */

/* 1st Example for Problem - Retrieve Full name of customers who had made atleast a single payment of more than 10$. */
SELECT first_name || ' ' || last_name FROM customer AS c
WHERE EXISTS(SELECT * FROM payment AS p WHERE amount > 10 AND p.customer_id = c.customer_id);
-- Helps in elimination of Join.


/* 2nd Example for Problem (using Multiple Approach)- Retrieve Records of Film title of rents, where return date is 29th May 2005. */

---- 1st Approach through "SUBQUERY" with "IN", (Helps in elimination of 1 JOIN) ----
--# Take Note at how SUBQUERY at different places result in same answer with different logic.
SELECT film.film_id, inventory_id, title
FROM film INNER JOIN inventory ON film.film_id = inventory.film_id
WHERE inventory_id IN (SELECT inventory_id FROM rental WHERE return_date BETWEEN '2005-05-29' AND '2005-05-30');

            /* --- OR ---- */

SELECT film_id,title FROM film WHERE film_id IN
(SELECT inventory.film_id FROM rental INNER JOIN inventory ON rental.inventory_id = inventory.inventory_id
WHERE return_date BETWEEN '2005-05-29' AND '2005-05-30') ORDER BY film_id;


---- 2nd Approach through "SUBQUERY" with "EXISTS" ----

SELECT inventory_id, title
FROM film INNER JOIN inventory ON film.film_id = inventory.film_id
WHERE EXISTS(SELECT inventory_id FROM rental WHERE return_date BETWEEN '2005-05-29' AND '2005-05-30');
--# Correcting above solution with addition of "Matching Condition". THIS IS IMPORTANT (for complete explanation, refer EXISTS from below)
SELECT inventory_id, title
FROM film INNER JOIN inventory ON film.film_id = inventory.film_id
WHERE EXISTS(SELECT inventory_id FROM rental WHERE return_date BETWEEN '2005-05-29' AND '2005-05-30'
                                                AND rental.inventory_id = inventory.inventory_id);
```

⇨ **CORRELATED SUBQUERY** results in mostly TABLE and are used mostly in WHERE/HAVING Clause with EXISTS, This is special case as in this it is not used as the Result but as a 'Boolean Function' to evaluate the Outer Query Records.

... WHERE EXISTS (subquery_Column_Table) ...;

INDEPENDENT TABLE

- We often have to refer to same 'Resultant Table' of some **'Complex Query'** (with Joins and conditions) **again and again**. Instead of calling this query again and again as a Starting Point, we can **assign** its result as either "**Virtual**" or "**Physical**" Table; And, then can directly access this specific table without writing query again as starting point.
- It's like assignment of output/resultant/retrieved table in Python or R.
- **Virtual Table** is like **Imaginary table**, which are not actually stored in the DATABASE, but either in the working memory or as a shorthand for query.
- We can Update and Alter existing Temporary Table {Virtual or Physical} also.
- There are mainly five ways to create Temporary Table. Namely-
  VIEW, CREATE TABLE AS, SELECT INTO, CTE,
- Main DIFFERENCES Between this are
  - VIEW only create Virtual, while CREATE and SELECT … INTO create both VIRTUAL AND PHYSICAL table.
  - We have more functionality with CREATE TABLE AS than SELECT … INTO.
  - CTEs are temporarily created Virtual Table Within the Main Query itself, which disappear after main_query executes.
- In 'CONCLUSION', Use this as per the situation -
  - Independent SUBQUERY can be used when we don't have to refer Temporary Table again and again for the FROM clause only.
  - VIEW are better option for creating VIRTUAL TABLE.
  - CREATE TABLE AS are better option for creating PHYSICAL TABLE as it provides more functionality.

VIEW

- It is the Clause for **assigning Query result into TABLE**.
- It will create "**Virtual Table**" only.
- View clause is placed at the starting of the statement.
- Syntax :=

  CREATE VIEW view_name AS
  QUERY;

- We can use "CREATE VIEW "**IF NOT EXISTS**"" to check whether the VIEW has already been created before. It allows us to bypass the Error if executed again and log it as a message.

  CREATE VIEW IF NOT EXISTS view_name AS QUERY;

```
CREATE VIEW film_category_view AS
SELECT name genre, title movie
FROM film JOIN film_category USING(film_id) JOIN category USING(category_id)
ORDER BY genre;

SELECT * FROM film_category_view;

--# Altering this view is very simple. CREATE OR REPLACE VIEW view_name AS
CREATE OR REPLACE VIEW film_category_view AS
SELECT name genre, title movie
FROM film JOIN film_category USING(film_id) JOIN category USING(category_id)
WHERE name = 'Action';

--# Renaming VIEW is also same as Physical Table.
ALTER VIEW film_category_view RENAME TO view_film_category;

--# Dropping VIEW is also same as Physical Table.
DROP VIEW IF EXISTS view_film_category;
```

CREATE TABLE AS

- It is the Clause for **assigning Query result into TABLE**.
- It will create either "**Physical Table**" or "**Virtual Table" with TEMP**.
- This clause is placed at the starting of the statement.
- Syntax :=

  CREATE [TEMP] TABLE new_table_name AS
  QUERY;

- Additional functionalities with CREATE TABLE AS. Note - These are not available in SELECT INTO Clause.
- We can use "CREATE TABLE "**IF NOT EXISTS**"" to check whether the Table has already been created before.
- It allows us to bypass the Error if executed again and log it as a message.

  CREATE TABLE IF NOT EXISTS new_table_name AS QUERY;
- The name and the datatype of the newly created table will be the same as of the column(s) selected in the query.
- E.g. if normal selection then same, if aggregation manipulation then integer, or array. To Overwrite the column name we can explicitly write the new_column_name_list by

  CREATE TABLE new_table_name(new_column_name_list) AS QUERY;

```sql
CREATE TEMP TABLE customer_address_virtual AS
SELECT first_name || ' ' || last_name full_name, address
FROM customer JOIN address USING(address_id)
ORDER BY full_name;

CREATE TABLE customer_address_physical AS
SELECT first_name || ' ' || last_name full_name, address
FROM customer JOIN address USING(address_id)
ORDER BY full_name;

SELECT * FROM customer_address_virtual;
SELECT * FROM customer_address_physical;
--# Modifying for this Tables are exactly as same as other Table in the Dataabase,
--  because now, they are also the Physical Table.
```

```sql
/*
- We can create Copy of Table using "CREATE TABLE AS", with variations-
  - TABLE STRUCURE with TABLE DATA.
  - TABLE STRUCTURE only meaning without TABLE DATA.
  - TABLE STRUCTRE with PARTIAL TABLE DATA.
*/

--# We can copy the Table Structure with its Data, as -
CREATE TEMP TABLE staff_copy AS
TABLE staff

--# We can copy just the Table Structure also without its Data, as -
CREATE TEMP TABLE staff_structure_copy AS
TABLE staff
WITH NO DATA;

--# We can copy Table with partial Data, as (Basically applying Predicate with WHERE) -
CREATE TEMP TABLE staff_partial_copy AS
SELECT * FROM staff WHERE staff_id = 1;
```

SELECT INTO

- It is the Clause for **assigning Query result into TABLE**.
- IT will create either "**Physical Table**" or "**Virtual Table" with TEMP**.
- INTO clause is placed between SELECTED column(s) list and FROM table_name WHERE predicate ...;
- We can use all the basic SQL Querying tools such as Clauses, Operators Functions, Joins for the SELECT INTO.

- **SELECT ... INTO [TEMP | TEMPORARY]** ==> also allows us to create temporary table which we can access further directly in other queries.
- SELECT ... INTO ==> Without the Optional Keyword TEMP or TEMPORARY, Physical Table will create instead of Virtual.
- Syntax :=

  SELECT column(s**) INTO [TEMP|TEMPORARY] temp_table_name** FROM ...;

```sql
SELECT name genre, title movie
INTO TEMP film_category_virtual
FROM film JOIN film_category USING(film_id) JOIN category USING(category_id)
ORDER BY genre;

SELECT name genre, title movie
INTO film_category_physical
FROM film JOIN film_category USING(film_id) JOIN category USING(category_id)
ORDER BY genre;

SELECT * FROM film_category_virtual;
SELECT * FROM film_category_physical;
--# Modifying for this Tables are exactly as same as other Table in the Database,
--  because now, they are also the Physical Table.

DROP TABLE IF EXISTS film_category_physical;
--# We will Delete the Physical table as we don't want to modify our DVDRENTAL DataBase.
```

CTEs

- 'CTEs or Common Table Expressions' are exactly like VIEWs, which create Virtual Table, but "Temporarily".
- Now the main difference between CTEs and VIEW are, that the **CTEs are created temporarily for the main query only**, and at the starting position within the main query itself. That is to say, **it disappears after the main query finishes execution**.
- Syntax :=

  WITH cte_1_name(column_list) AS (QUERY),
  cte_2_name(column_list) AS (QUERY),
  ...
  <MAIN_QUERY>; --# Main Query referencing CTE(s).

- It is optional to mention 'column_list' explicitly. We write this to overwrite the column_name from SELECT clause of cte_query.
- Now the **execution Step/Process** is such that -
  1 - the CTE mentioned by WITH Clause are executed first and creates the virtual table(s).
  2 - Main query reference these Virtual Table(s) and Execute itself.
  3 - Virtual Table created by CTE disappears after the query execution.
  4 - Data Retrieved is displayed to the user.

- Note that through this steps, Logically, cte_query cannot refer or be related to the Main query statement. Meaning it is Independent.
- This is also why and how it different from Nested Subquery.

```sql
/*
-- Situation - Display each movie's genre and its Cast.
-- What I've Done is basically I've created two temorary CTE table with WITH  -
     One for retrieving 'Genre' from simple join of film with category.
     And second for retrieving 'CAST' from simple join of film with actor
     (by aggregating all the actors in single cell as array);
     And, then I've joined these two tables in main query with film_id.
*/

WITH
cte_film_category(film_id, movie, genre) AS
(SELECT film_id, title, name
 FROM film JOIN film_category USING(film_id) JOIN category USING(category_id)),
cte_film_actor(film_id, movie, actors) AS
(SELECT film_id, title, ARRAY_AGG(first_name::varchar || ' ' || last_name::varchar)
 FROM film JOIN film_actor USING(film_id) JOIN actor USING(actor_id) GROUP BY film_id,title)
SELECT f_c.film_id, f_c.movie,f_c.genre,f_a.actors
FROM cte_film_category f_c JOIN cte_film_actor f_a USING(film_id)
ORDER BY film_id;
```

RECURSIVE CTE

- If we want to **use the one's own query result within the query call**, we can do so using recursive CTE.
- Recursive CTE use keyword "**WITH CTE**" which consist of two statements.
  First, **Initial Statement**, which generates the Initial data or Table.
  Second, **Recursive Statement**, which calls its own query output table.
  Lastly, we append the records from each query execution to its self-output table.
- Thus, it allows us to work even further (again and again) on the result that it itself has already produced.

General **Flow** and logic of the WITH RECURSIVE

- It needs an **initial statement** to generate the first sets of records (on which we want to work further on)
- Second, it needs an **recursive statement,** which will call its own "previous run output" and further executes on it. (Note, recursive statement has access to those records only which are output of the last iteration and not the whole table.)
- Recursive CTE will **end** either 'by itself, if the recursive statement **hits null** or 'if we add any **stoppage condition** in the WHERE clause of the recursive statement'.
- Lastly, it needs an "**UNION ALL**" to append the records retrieved in the recursive queries in one single table.

- It's basic structure and syntax is

```
WITH RECURSIVE table_name(column(s)) AS (
        <initial statement>
        UNION ALL
        <recursive statement>
) Main_query;
```

- Keep note, that the column structure should be compatible in the initial statement and the recursive statement

```sql
-- Basic Recursive statement example
WITH RECURSIVE increment(num) AS (
    SELECT 1
    UNION ALL
    SELECT num+1 FROM increment WHERE num < 5
) SELECT * FROM increment;
```

```sql
/*
Exercise -
https://pgexercises.com/questions/recursive/getupward.html
Find the upward recommendation chain for member ID 27: that is, the member who recommended them,
and the member who recommended that member, and so on. Return member ID, first name, and surname.
Order by descending member id.
*/

WITH RECURSIVE recommendation(memid, recommender, firstname, surname, recommendedby) AS (
    SELECT m1.memid, m1.recommendedby, m2.firstname, m2.surname, m2.recommendedby
    FROM cd.members m1 JOIN cd.members m2 ON m1.recommendedby = m2.memid
    WHERE m1.memid=27

    UNION ALL

    SELECT r.recommender, r.recommendedby, m3.firstname, m3.surname, m3.recommendedby
    FROM recommendation r JOIN cd.members m3 ON r.recommendedby = m3.memid
)
SELECT recommender, firstname, surname FROM recommendation;
```

SUBQUERY & INDEPENDENT TABLE OVERVIEW

Many a times we need to make use of complex query to perform complex analysis and manipulation. Assignment of 'Table Result' in another 'Temporary Table' help us by simplifying the complex query into simple query call. That is one query result is used in another query, rather than writing this query again and again as a subquery in many queries to make it complex.

- ⇨ SUBQUERY = Subquery are used to perform complex query in some sensible way. Sometimes, we cannot perform queries with simple query statement. Often, we need to make use of Subquery, to use its result in the Main query. It can be Independent or Correlated. Correlated Subquery means columns in subquery have some linking to columns in main query. In this case record is executed row by row with main query to derive the result and finally execute the main query.
- ⇨ VIEWS = View are Virtual Table created for storing the Query Result in some Temporary Form to use its result in many other queries.
- ⇨ CREATE TABLE AS = These are used to create Physical table rather than Virtual Table of some query result. It provides many functionalities to create table.
- ⇨ SELECT INTO = These are same as CREATE TABLE AS, i.e., to create physical table but with less functionalities than CREATE TABLE AS.
- ⇨ CTEs = CTEs are special type of Temporary Result of table. These are written as subquery but are used exactly as same as VIEW. These are the virtual tables written in the query itself to make use of it, which disappears after the execution of query, unlike VIEWs.

MODIFYING DATA

THEORETICAL CONCEPTS

- ⇨ **DATATYPE**
- ⇨ **PRIMARY KEY & FOREIGN KEY**
- ⇨ **CONSTRAINT**

DATA MODIFY keywords, clauses, statements

- ⇨ **CREATE**
- ⇨ **INSERT**
- ⇨ **UPDATE**
- ⇨ **DELETE**
- ⇨ **DROP/TRUNCATE**
- ⇨ **ALTER**

```
/* --- 6. CREATION and MODIFYING TABLE --- */
THEORETICAL CONCEPTS
- DATA TYPE
- PRIMARY and FOREIGN KEY
- CONSTRAINTS
STATEMENTS
- CREATE
- INSERT
- UPDATE
- DELETE
- DROP/TRUNCATE
- ALTER
```

➢ THEORETICAL CONCEPTS

DATATYPES

- SQL supports following Datatypes -
  Common Ones
  - Boolean - True or false
  - Character - char, varchar or text
  - Numeric - integer and floating-point number
  - Temporal - Data, Time, Timestamp, Interval
  Non Common Ones -
  - UUID - Universally Unique Identifiers - Algorithmically Unique Code to identify particular Row.
  - Arrays - Stores an Array of Strings, Numbers, etc.
  - JSON
  - HStore Key-Value Pair
  - Special Types - Network Address AND Geometric Data.

- Best practice to decide which datatype to used is "Always take a look online or at Documentation".
- Select the most convenient Primary data types and in those take a look at Doc as there are many secondary datatypes.

- Refer to Documentation to see limitations of Data Types := postgresql.org/docs/current/datatype.html
- E.g. - For Phone number - Either Integer OR Char, and if integer then what type of integer, and if Char then what type of Char.

PRIMARY KEY and FOREIGN KEY

- **Primary Keys** are the fundamental concept for the table.
- Primary keys are the column which store the **unique identifier** or **ID** for the Rows of the particular Table.
- One can say it's the **address** for the particular Record or particular Row or particular Data Instance.
- Every Table in the particular Database will necessarily have its own unique Primary key.
- These Primary key are also useful for forming Relation or Schema between tables in the Particular Database.
- Primary key must be integer and NOT NULL for the table. It is mentioned as [PK] in the PostgreSQL pgadmin's constraint.

- **Foreign keys** are the fields which are the **unique identifier column in a table for the another table**. That is they uniquely identifies a row in the another table.
- A Foreign Key is defined in a table as the reference to the primary key of the other table. i.e., they refer to the primary key of other table.
- In a Nutshell, they are Primary keys of some other table.
- The table that contains the foreign key are called "**Child Table**" or "**Referencing Table**".
- The table they (foreign key) refers to are called "**Parent Table**" or "**Referenced Table**".
- A table can have multiple foreign keys depending on its relationships with other tables.
- We can specify Primary key OR attaching a Foreign Key relation to other table USING "CONSTRAINTS".

CONSTRAINTS

- Constraint **enforced the rules** on table "Columns" or whole "Table" in general.
- Constraint on Columns are called "Column Constraints". Constraints on entire table are called "Table Constraints".
- They are used to **prevent inconsistent or invalid data** to be entered in the databases table.
- And use to make sure that the tables are consistent.
- Thus it ensures the reliability, accuracy and consistency of the data in database.

- Most Common "**COLUMN CONSTRAINTS**" Used:
  - **NOT NULL** - Ensures that column cannot have a NULL value.
  - **UNIQUE** - Ensures that every value in the column are DISTINCT.
  - **PRIMARY KEY** - uniquely identifies the row/record in the table.
  - **FOREIGN KEY** - uniquely identifies the row/record in another table.
  - **CHECK** - ensures that every values in the column must satisfy certain custom conditions (which are added additionally with CHECK).
  - **REFERENCES** - to constraint the value in the column which must exist in a column of another table.
  - **EXCLUSION** - ensure that if any two rows are compared on specified column or expression using the specified operator, not all of these comparisons will return TRUE.

- Most Common "**TABLE CONSTRAINTS**" Used:
- - **CHECK(condition)** - To check condition when inserting or updating data in a table.
- - **UNIQUE(column_list)** - Values inside the listed column HAVE TO be unique within all the listed columns.
- - **PRIMARY KEY(column_list)** - Allows us to define the PRIMARY KEY consisting of a MULTIPLE COLUMNS.

➢ DATA MODIFY Query Statements

CREATE

- CREATE statement allows us to create TABLE.
- It creates the Table with "Table Structure" and not with the actual data.
- FULL GENERAL SYNTAX :-

CREATE TABLE table_name (
column_name_1 DATATYPE column_constraints,
column_name_2 DATATYPE column_constraints,
column_name_3 DATATYPE column_constraints,
...
table_constraints
)

```
CREATE TABLE account(
    user_id SERIAL PRIMARY KEY,
    username VARCHAR(50) NOT NULL UNIQUE,
    password VARCHAR(50) NOT NULL,
    email VARCHAR(250) NOT NULL UNIQUE,
    created_on TIMESTAMP NOT NULL,
    last_login TIMESTAMP
);


CREATE TABLE JOB (
    job_id SERIAL PRIMARY KEY,
    job_name VARCHAR(250) NOT NULL UNIQUE
);


CREATE TABLE account_job(
    account_id INTEGER REFERENCES account(user_id),
    job_id INTEGER REFERENCES job(job_id),
    hired_date TIMESTAMP
);
```

INSERT

- CREATE statement only generates the table structure and we cannot **add data** with CREATE statement.
- For that we need to use another statement called INSERT.
- **INSERT** allows us to **add data/record/rows into the TABLE**.
- **VALUES** is used to **construct a row of data**. General Syntax :=

INSERT INTO table_name (column_1, column_2, ...)
VALUES
(value1, value2, ...),
(value1, value2, ...),
...;

- If we are providing data for every column, then specifying column name in the INSERT statement is Optional.
- Also, we must pass the values in row construct AS per the order of column mention in the INSERT phrase. Though, we can mention column in any order in the INSERT phrase.
- 2 point to bear in mind. -
  - Inserted rows values must match up to the column for the table. and it should follow constraints.
  - In short, Rows added must be compatible with the table structure.
  - SERIAL column do not need to be provided values.

- **INSERT ... SELECT** allows us to **add data**/record/rows into a table **FROM another Table**. General Syntax:=

INSERT INTO table_name (column_1,column_2,...)
SELECT column_1,column_2,...
FROM another_table WHERE condition_if_any;

```
INSERT INTO account(username,password,email,created_on)
VALUES
('Ashish','password','shisha@gmail.com',CURRENT_TIMESTAMP);
--# Verify =>                      SELECT * FROM account
-- See that, we did not mention and added SERIAL column and unique identifier.


INSERT INTO job(job_name)
VALUES
('Data Scientist'),
('Instructor/Teacher');
--# Verify =>                      SELECT * FROM job


INSERT INTO account_job(account_id,job_id,hired_date)
VALUES
(1,2,CURRENT_DATE),
(2,1,'2024-01-01');
--# Verify =>                      SELECT * FROM account_job
--# If we try to insert record with foreign key value which does not exists in the Referenced Table,
--  it will results in Foreign Key Constraint Violation ERROR.
```

⇨ CHECK constraint

- CHECK Constraint allows us to create more **CUSTOMIZED CONSTRAINTS** that column values must adhere to.
- E.g. - Custom condition specifying 'Integer value range' or 'must be positive', etc.
  - Category value (numeric or text) must be from within the specified list.
- Syntax := CHECK constraint appears as "**CHECK (conditions)**" at the constraint place after the datatype in Column phrase.
- E.g. - age SMALLINT NOT NULL CHECK (age > 21),
  - parent_age SMALLINT CHECK (parent_age > age),
  - gender CHAR(1) NOT NULL CHECK (gender IN ('M','F','T'));

```
CREATE TABLE employees(
    emp_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    gender CHAR(1) NOT NULL CHECK (gender IN ('M','F','T')),
    DOB DATE CHECK (DOB > '1900-01-01'),
    hired_date DATE CHECK (hired_date > DOB),
    salary INTEGER CHECK (salary > 0)
);

INSERT INTO employees(first_name, last_name, gender, DOB, hired_date, salary)
VALUES
    --# ('Jose','Portilla', 'M','1857-11-03','2010-01-01',100),   --# Will give error "employees_dob_check"
        ('Jose','Portilla', 'M','1996-11-03','2010-01-01',100),
    --# ('Sammy','Smith', 'M','1996-11-03','2010-01-01',-120),  --# Will give error "employees_salary_check"
        ('Sammy','Smith', 'M','1996-11-03','2010-01-01', 120),
    --# ('Karen','Pandey', 'O','1996-11-03','2010-01-01',50),  --# Will give error "employees_gender_check"
        ('Sammy','Pandey', 'F','1996-11-03','2010-01-01', 50);
```

UPDATE

- UPDATE allow for changing values or modifying values in the table columns. General Syntax :=

UPDATE table_name
SET
column_1 = value_1,
column_2 = value_2,

```
    ...                 --# Whatever column or attributes we have to change.
WHERE
    condition;          --# And for Whatever Rows or Records we have to change.
```

- If we want to reset every record, remove the WHERE Clause in UPDATE statement.

```
UPDATE account
SET
    last_login = CURRENT_TIMESTAMP
WHERE last_login is NULL;
```

NOTE

- We can **UPDATE** 'multiple rows' of a column **based on 'another list or column'** as stated below. But for this, bear 2 points in mind
  - list size should match on both side of "=" of SET. That means, it should be compatible.
  - values that we are updating in a column must follow all the constraints of a column.
- Update values (for all rows) **based on another column value**. Example :=

```
UPDATE account
SET last_login = created_on;
```

- Update values (for conditional rows) **based on another table's column value**. This is also known as "UPDATE Join" though Join is not used in syntax. Syntax :=

```
UPDATE table_A
SET table_A.col = table_B.some_col
FROM table_B
WHERE table_A.id = Table_b.id;
```

- FROM clause in UPDATE statement allows us to generate and retrieve values for use in the SET clause.
- We can also return the selected columns of 'Affected or Updated Rows' in UPDATE statement using **RETURNING** keyword. This is very useful as we don't have to run another SELECT statement to verify.

```
UPDATE account
SET last_login = created_on
RETURNING account_id, username, last_login;
```

```
UPDATE account
SET last_login = created_on;

UPDATE account
SET created_on = account_job.hired_date
FROM account_job
WHERE account.user_id = account_job.account_id;

UPDATE account
SET last_login = CURRENT_TIMESTAMP
RETURNING username, created_on, last_login;
```

DELETE

- DELETE clause allows us to remove ROW or Records from the Table.
- We can Delete **all ROWS**/RECORDS by not specifying WHERE condition. Syntax :=

```
DELETE FROM table_name;
```

- We can Delete **specific ROWS**/RECORDS by specifiying filtering condition with WHERE. Syntax :=

  DELETE FROM table_name
  WHERE condition(s);

- We can DELETE ROWS/RECORDS **based on their existence in another table**. This is same as INSERT Join or UPDATE join without the join keyword. Syntax :=

  DELETE FROM table_A
  USING table_B
  WHERE table_A.id = table_B.id;

- Similar to UPDATE command, we can use **RETURNING** keyword for Returning ROWS that were removed.

  DELETE FROM table_name
  WHERE conditions
  RETURNING;

```
DELETE FROM job
WHERE job_name = 'Being Karen'
RETURNING *;
```

DROP/TRUNCATE

- DROP allows for complete removal of an **Object** in a Database. DROP OBJECT object_name;
- By Objects, we meant - **Column, Table, Views**
- In PostgreSQL, it removes column and its index, constraints. But it doesn't remove dependencies based on this column. Like views, triggers, or stored procedures. To remove this dependencies we need the additional **CASCADE** clause.
- Syntax :=

  DROP TABLE table_name;
  DROP VIEW view_name;

- Dropping Column, Syntax :=

  ALTER TABLE table_name
  DROP COLUMN column_name --# Just to remove column and its content.
  --or--
  DROP COLUMN column_name CASCADE --# To remove column with its dependencies.
  --or--
  DROP COLUMN IF EXISTS col_name --# Check for Existence to avoid error.
  --or--
  DROP COLUMN col_1 --# To remove Multiple Columns.
  DROP COLUMN col_2

TRUNCATE

- TRUNCATE is like DROP that Deletes all the DATA inside the Table, but the difference here is that it does it does not Delete the Whole Table
- That is to say, it **preserves the Table and Table structure** for further use. But Yes all the data inside it will get eliminated.

  TRUNCATE TABLE table_name;

```
ALTER TABLE new_info
DROP COLUMN IF EXISTS people;
```

ALTER

- ALTER Clause allows us to **modify** or bring change to an existing "**TABLE STRUCTURE**", such as -
  - Rename Table.
  - Adding Column, Dropping Column, or Renaming Column.
  - Changing a Column's DATA TYPE.
  - SET or DROP column CONSTRAINTS (NOT NULL, UNIQUE, DEFAULT, KEYS)
  - Add or Modify CHECK constraints.

- **General Syntax** :=

  ALTER TABLE table_name
  RENAME TO new_table_name; --# RENNAME Table

-

- Altering Table's Column Structure

  ALTER TABLE table_name
  ALTER COLUMN column_name
  "Action";

  ---- "Action" comprises of ----
  ADD COLUMN column_name DATATYPE; --# ADD Column
  DROP COLUMN column_name; --# DROP Column
  RENAME COLUMN col_name TO new_col_name; --# RENAME Column
  SET constraint; --# SET column Constraint
  DROP Constraint; --# DROP Column Constraint
  ADD Constraint; --# ADD Column Constraint
  TYPE datatype; --# Changing Column's DATATYPE

```sql
/* --- Let's create a Demo Table for ALTERing. --- */
CREATE TABLE info(
    info_id SERIAL PRIMARY KEY,
    title VARCHAR(50) NOT NULL,
    person VARCHAR(50) NOT NULL UNIQUE
);
--# View the Demo Table.
SELECT * FROM info;    SELECT * FROM new_info;

/* --- RENAMING Table_name --- */
ALTER TABLE info
RENAME TO new_info;

/* --- RENAMING column_name --- */
ALTER TABLE new_info
RENAME COLUMN person to people;

/* --- ADD column --- */
ALTER TABLE new_info
ADD COLUMN age INTEGER;

/* --- DROP column --- */
ALTER TABLE new_info
DROP COLUMN age;

/* --- Modify DATATYPE of Column --- */
ALTER TABLE new_info
ALTER COLUMN age TYPE VARCHAR(2);
```

```sql
/* --- SET | DROP "NOT NULL" Constraint on Column --- */
ALTER TABLE info
ALTER COLUMN person
DROP NOT NULL    --# Similarly, SET NOT NULL

/* --- ADD UNIQUE Constraint on Column --- */
ALTER TABLE info
ADD UNIQUE(title)

/* --- SET | DROP DEFAULT Value of Column --- */
ALTER TABLE new_info
ALTER COLUMN age
SET DEFAULT 18  --# DROP Default
```

MODIFYING DATA OVERVIEW

MODIFYING Statement are used to create, modify and manage the Data in the Table. Thing to bear in mind for modifying Table is, we should make note of Table Structure (Column and its Constraints or Datatypes) while modifying so that, it is reliable, consistent and compatible.

- ⇨ CREATE = Create are used to create table in a Database with the table structure, i.e., Column datatypes, column constraints, table constraints and its primary key and foreign key referencing.
- ⇨ INSERT = Insert statement allows us to insert the data records in the newly created or already existing table.
- ⇨ UPDATE = Update statement allows us to update the Column Values.
- ⇨ DELETE = Delete allows us to delete records from the table either based on condition or whole records
- ⇨ DROP/TRUNCATE = DROP allows us to Delete columns or any other object like Table, View, Schema, etc. TRUNCATE allows to delete table's data but retain its structure for its further use.
- ⇨ ALTER = Alter Table allows us to alter the Table structure like altering its column, renaming objects name within the table, or changing its datatype, constraints.
