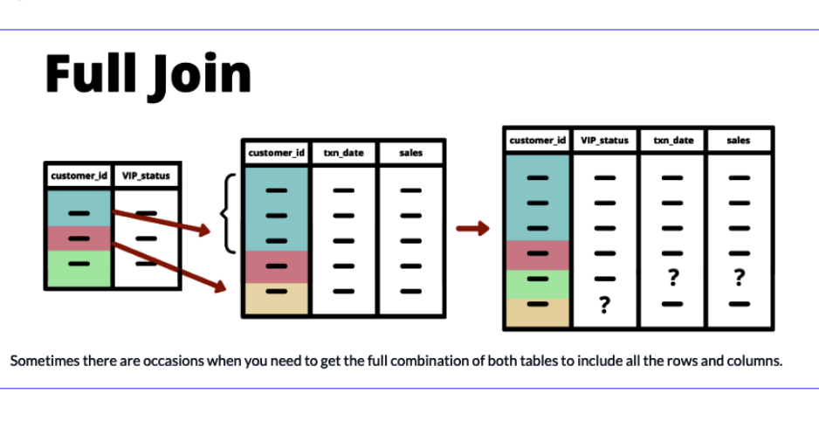
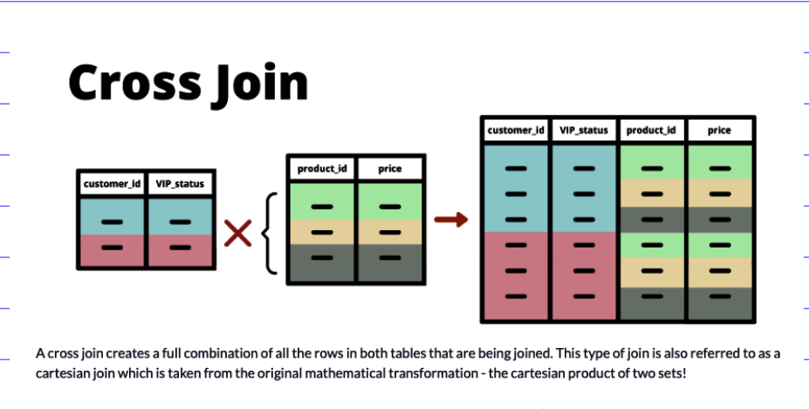
→ Temporary table + CTE

```
DROP TABLE IF EXISTS <table_name>;
CREATE TEMP TABLE <table_name> AS

    WITH <cte_name> (col_1, col2, ..., coln) AS (

        VALUES
        (   ,   ,   ,      ),
        (   ,   ,   ,      ),
        (   ,   ,   ,      )

    )
    select * from <cte_name>;
```



## Full Join

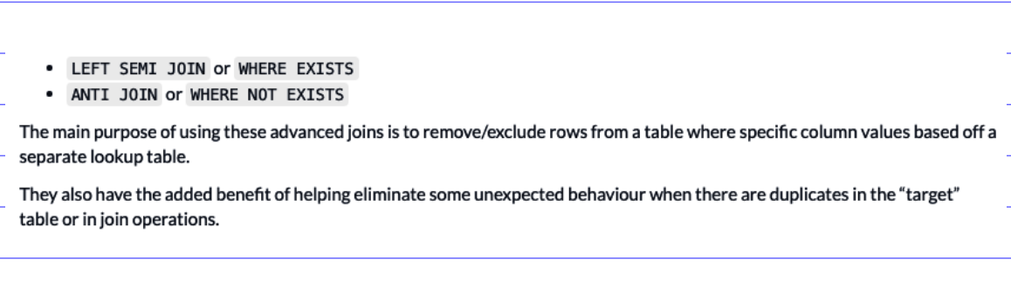
Sometimes there are occasions when you need to get the full combination of both tables to include all the rows and columns.



## Cross Join

A cross join creates a full combination of all the rows in both tables that are being joined. This type of join is also referred to as a cartesian join which is taken from the original mathematical transformation - the cartesian product of two sets!

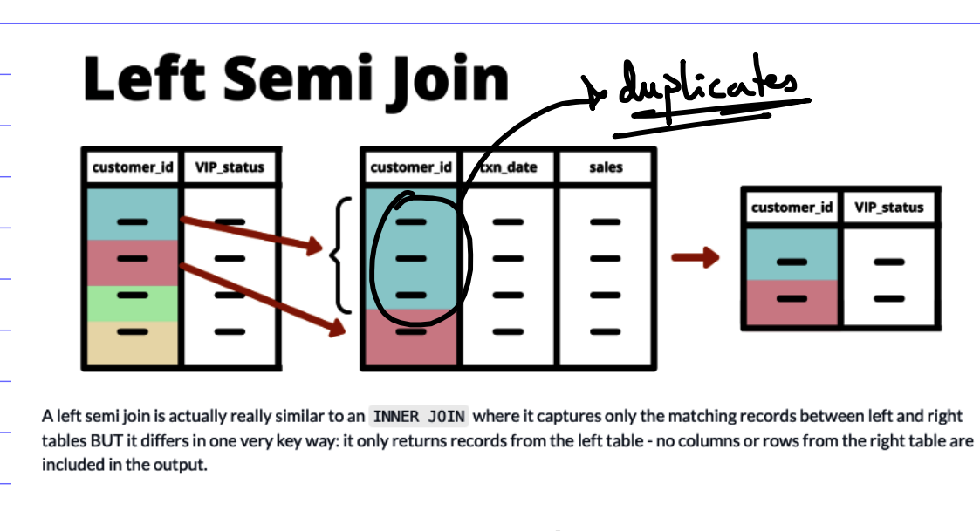
→ FROM t1 CROSS JOIN t2 / FROM t1, t2



- `LEFT SEMI JOIN` or `WHERE EXISTS`
- `ANTI JOIN` or `WHERE NOT EXISTS`

The main purpose of using these advanced joins is to remove/exclude rows from a table where specific column values based off a separate lookup table.

They also have the added benefit of helping eliminate some unexpected behaviour when there are duplicates in the "target" table or in join operations.



## Left Semi Join

A left semi join is actually really similar to an `INNER JOIN` where it captures only the matching records between left and right tables BUT it differs in one very key way: it only returns records from the left table - no columns or rows from the right table are included in the output.

→ Target table acts as a filter

Syntax:

ORDER OF TABLES IMPORTANT

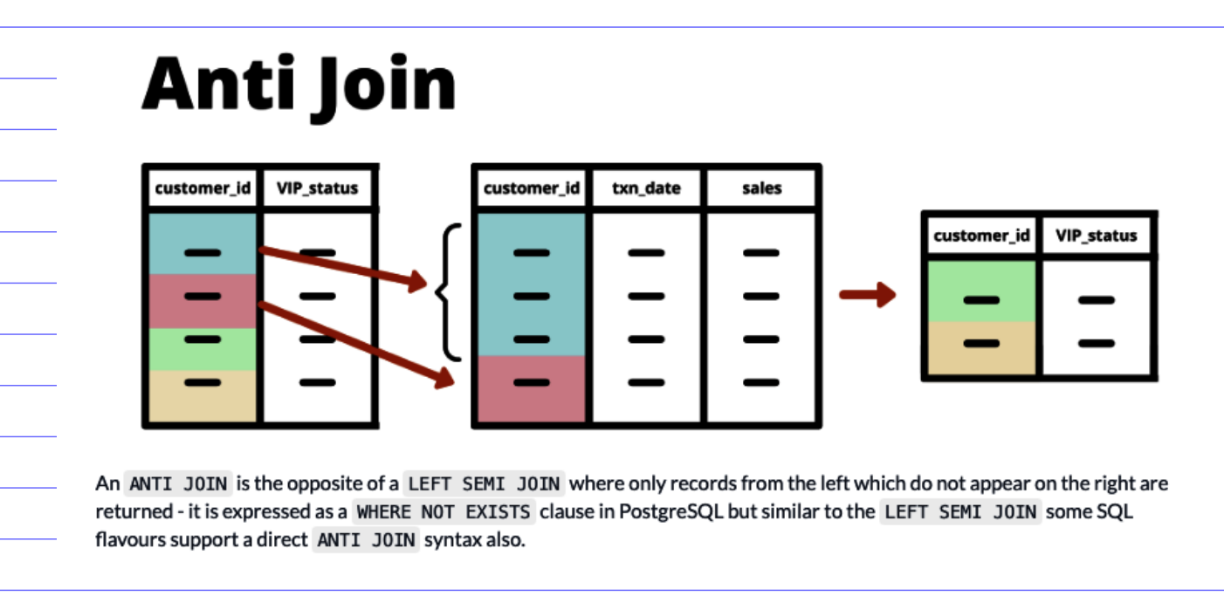
```
SELECT                              SELECT
  ~~~                                 ~~~
FROM <base_table>                   FROM <base_table>
WHERE EXISTS (                      LEFT SEMI JOIN <target_table>
  SELECT 1                          ON ~~~
  FROM <target_table>
  WHERE ~~~
);
```

Doesn't return any data

→ Depending on the columns chosen Distinct may not remove duplicates

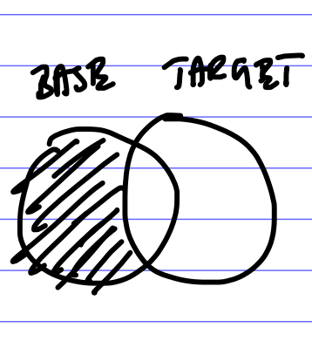
```
SELECT DISTINCT                     SELECT DISTINCT
  base_table.~~                       base_table.~~
FROM <base_table>                   FROM <base_table>
LEFT JOIN <target_tables>           INNER JOIN <target_table>
ON ~~~                              ON ~~~
WHERE target_table.~~ IS NOT NULL ;
```

INEFFICIENT WHEN WE WANT ONLY RESULTS FROM BASE TABLE.



## Anti Join

An `ANTI JOIN` is the opposite of a `LEFT SEMI JOIN` where only records from the left which do not appear on the right are returned - it is expressed as a `WHERE NOT EXISTS` clause in PostgreSQL but similar to the `LEFT SEMI JOIN` some SQL flavours support a direct `ANTI JOIN` syntax also.



Syntax:

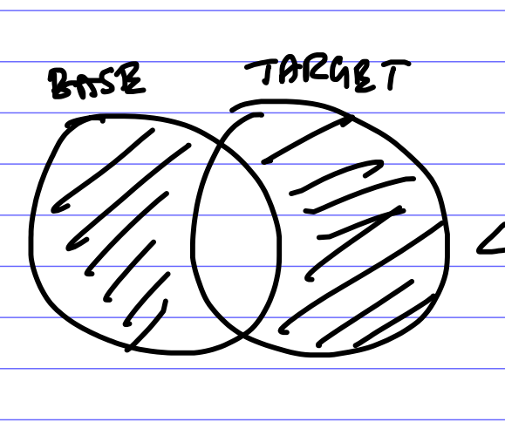
```
SELECT                              SELECT
  ~~~                                 ~~~
FROM <base_table>                   FROM <base_table>
WHERE NOT EXISTS (                  ANTI JOIN <target_table>
  SELECT 1                          ON ~~~
  FROM <target_table>
  WHERE ~~~
);
```

OR ← MORE EFFICIENT

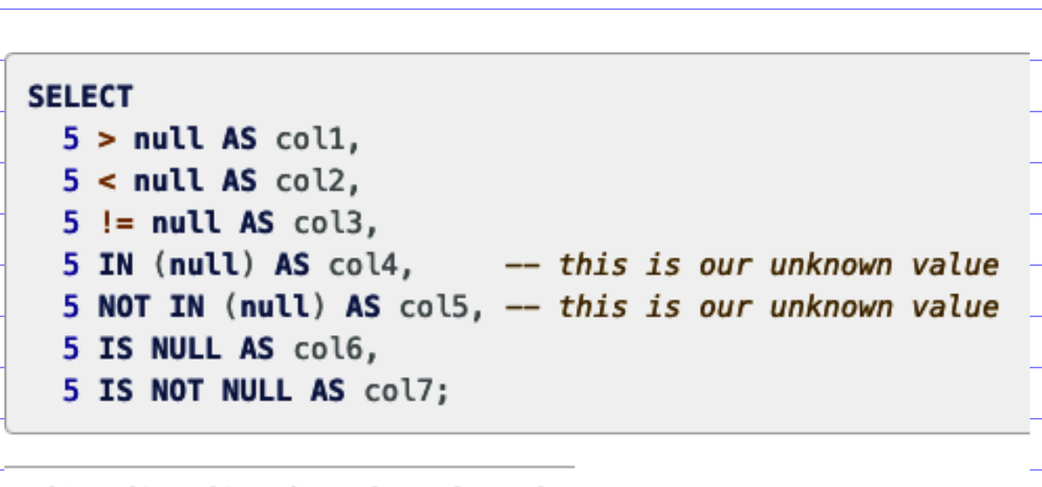
```
SELECT
  ~~
FROM <base_table>
LEFT JOIN <target_table>
ON ~~~
WHERE <target_table>.~~ IS NULL;
```

* If we are joining tables which have NULL in the JOINING Column THE RESULTS DON'T ACT AS EXPECTED.

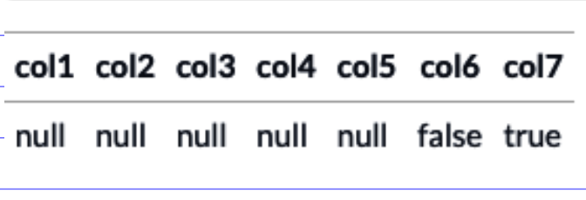
NULL ≠ NULL



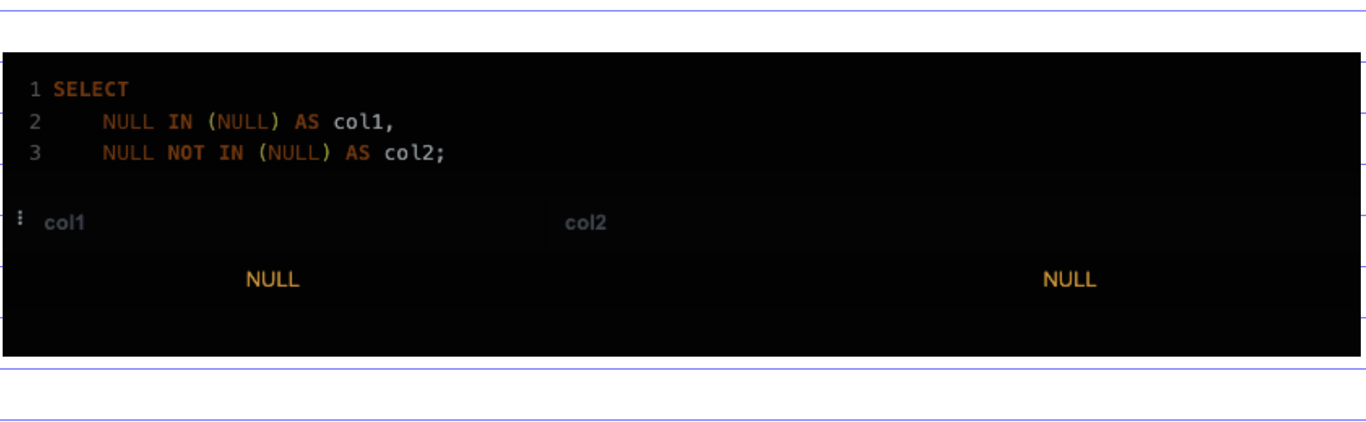
← Symmetric Difference



```
SELECT
  5 > null AS col1,
  5 < null AS col2,
  5 != null AS col3,
  5 IN (null) AS col4,      -- this is our unknown value
  5 NOT IN (null) AS col5, -- this is our unknown value
  5 IS NULL AS col6,
  5 IS NOT NULL AS col7;
```

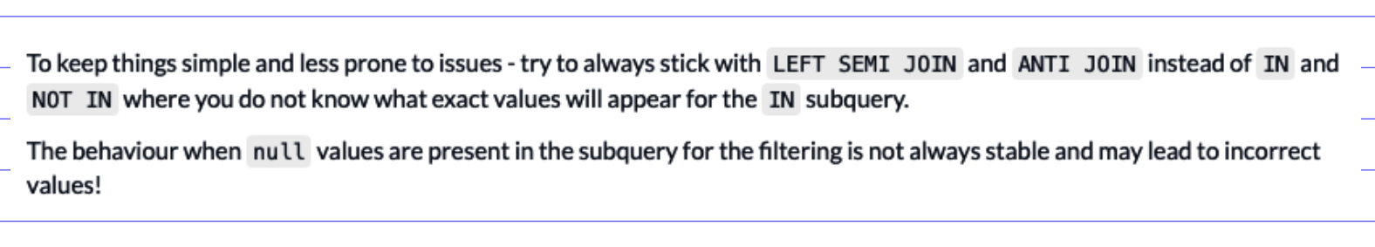


| col1 | col2 | col3 | col4 | col5 | col6 | col7 |
|---|---|---|---|---|---|---|
| null | null | null | null | null | false | true |



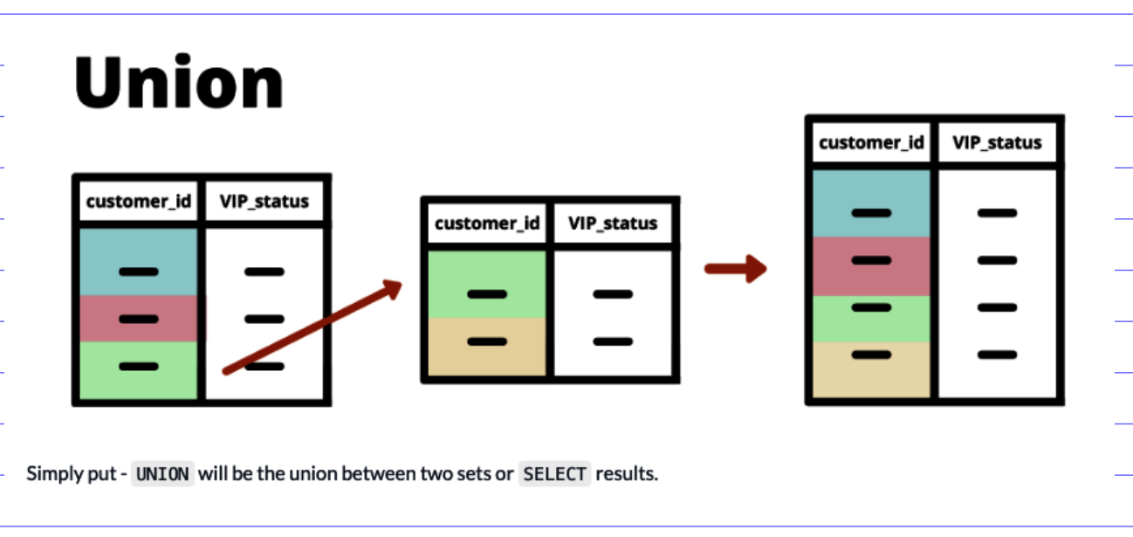
```
1 SELECT
2     NULL IN (NULL) AS col1,
3     NULL NOT IN (NULL) AS col2;
```

```
col1          col2
NULL          NULL
```



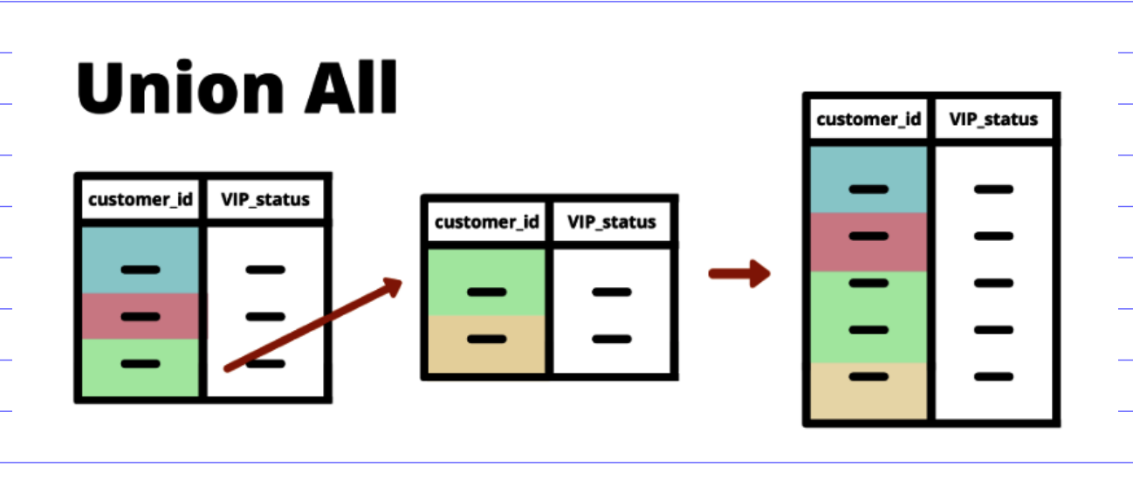
To keep things simple and less prone to issues - try to always stick with `LEFT SEMI JOIN` and `ANTI JOIN` instead of `IN` and `NOT IN` where you do not know what exact values will appear for the `IN` subquery.

The behaviour when `null` values are present in the subquery for the filtering is not always stable and may lead to incorrect values!

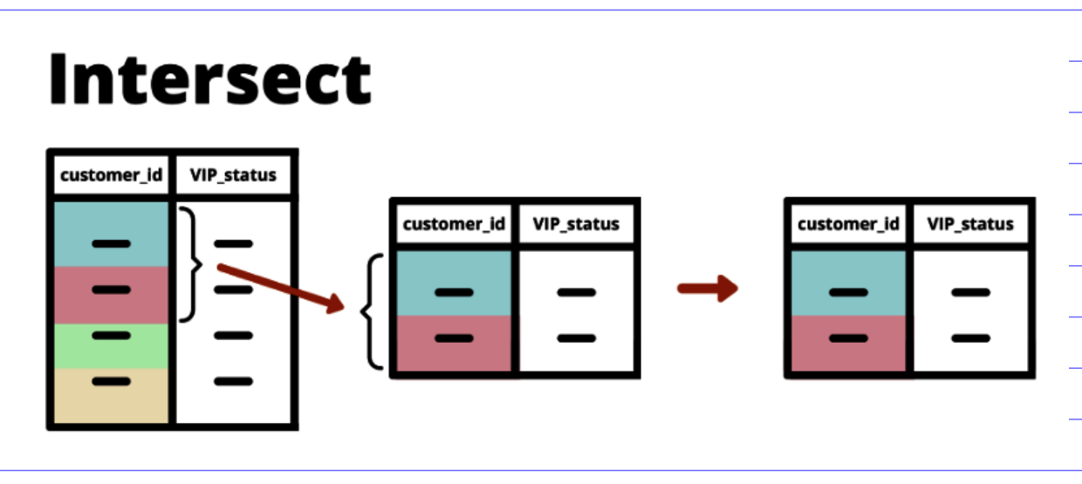


## Union

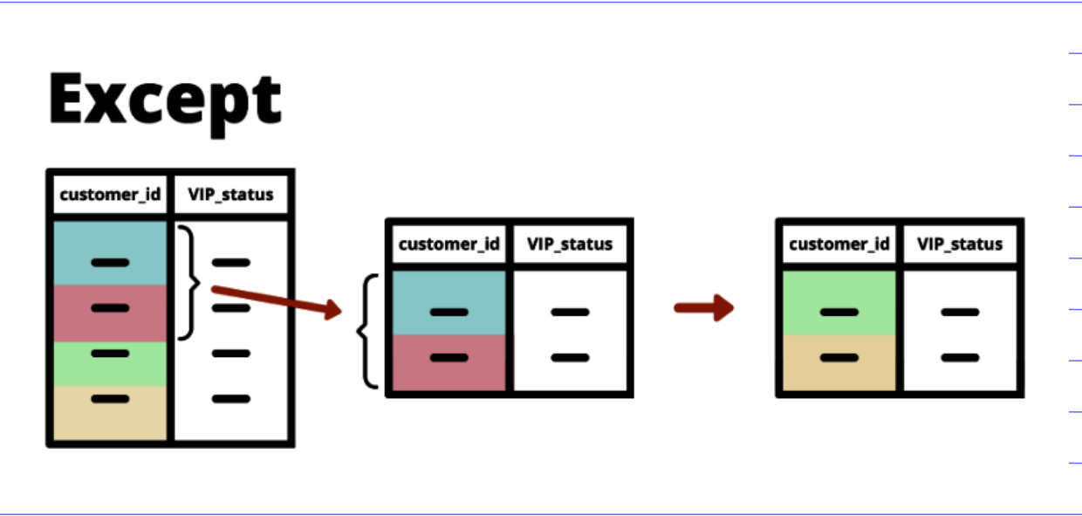
Simply put - `UNION` will be the union between two sets or `SELECT` results.



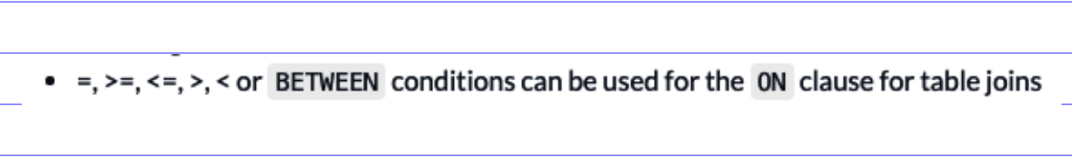
## Union All



## Intersect



## Except



- =, >=, <=, >, < or `BETWEEN` conditions can be used for the `ON` clause for table joins

```
→ UPDATE <table>
  SET
    <col> = ____,
    <col> = ____
  WHERE ~~~~
  RETURNING *
```

→ shows all updated rows as query output

→ Familiarizing with new database using the information_schema

- we can check datatypes using information_schema.columns table

```
select
    table_name
    column_name
    data_type
from information_schema.columns
where table_name in ( ~~~~ );
```

High lvl overview of the dataset we will be working with

```
select
  schema_name,
  table_name,
  column_name
from information_schema.columns
where column_name ILIKE '____';
```

→ Checking for missing values in multiple columns?

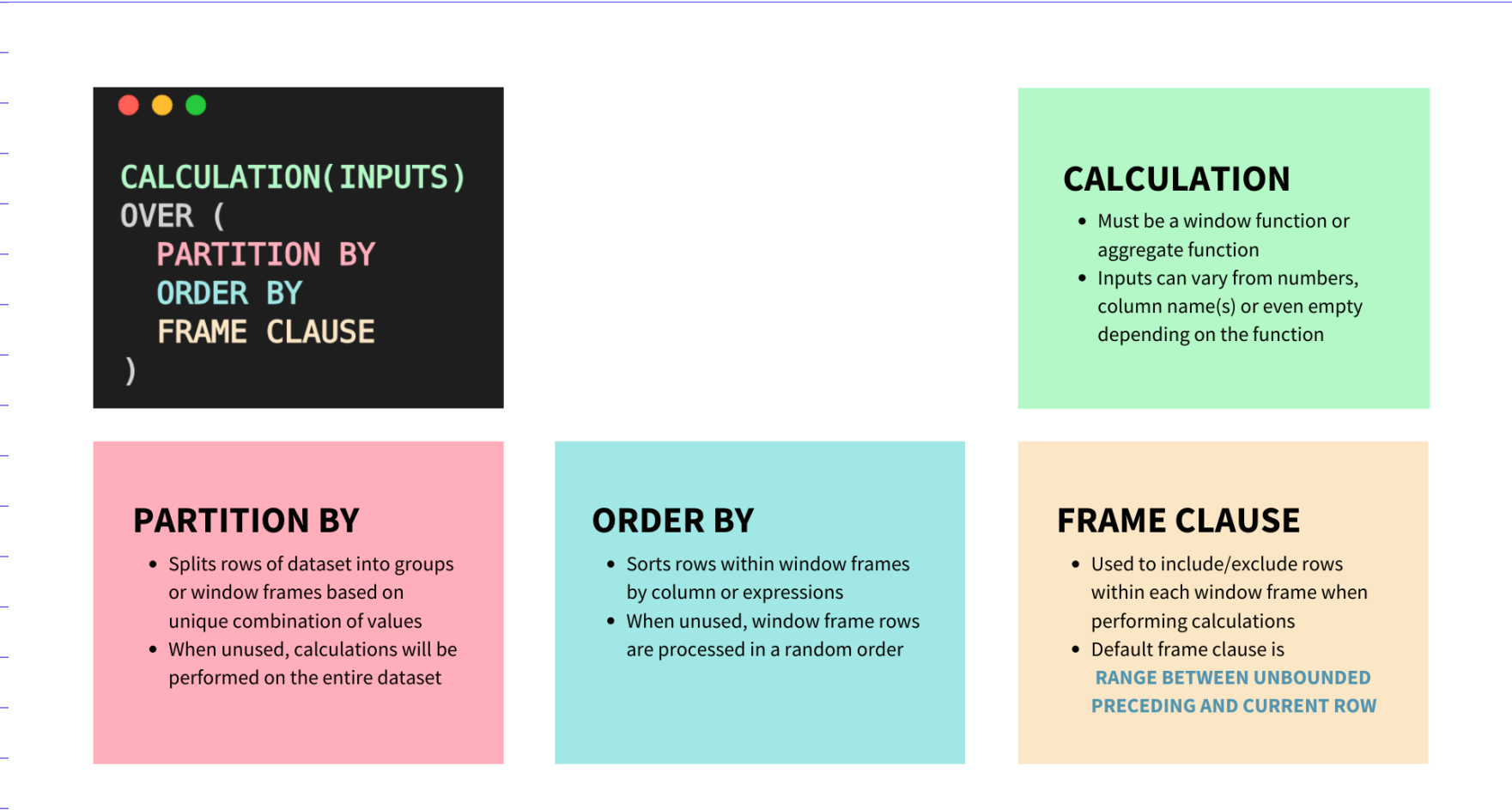
```
select *
from <table>
where (col_1 + col_2 + ... + col_n) IS NULL;
```

↳ Propagation of nulls

1 + null → null

* When dealing with time series based data & we encounter missing values, use the day before NEVER the day after.

DO NOT USE FUTURE DATA



```
CALCULATION(INPUTS)
OVER (
  PARTITION BY
  ORDER BY
  FRAME CLAUSE
)
```

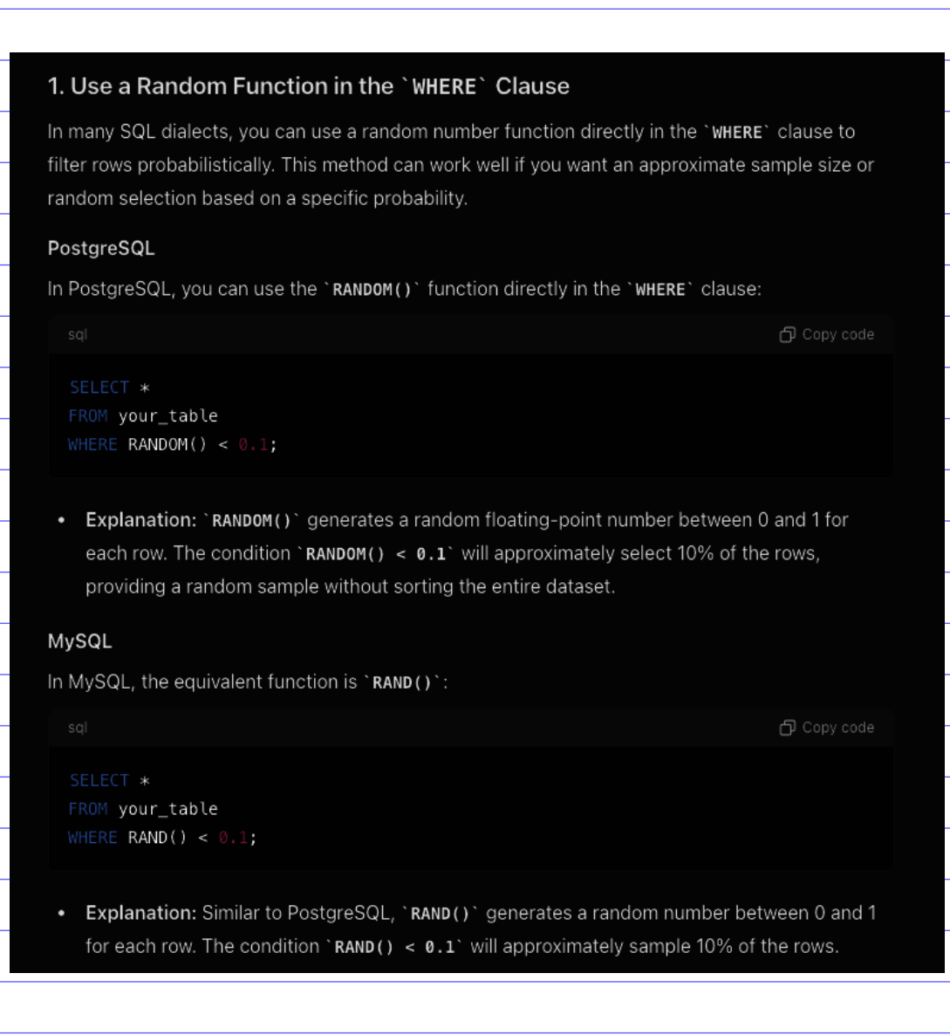
**CALCULATION**
- Must be a window function or aggregate function
- Inputs can vary from numbers, column name(s) or even empty depending on the function

**PARTITION BY**
- Splits rows of dataset into groups or window frames based on unique combination of values
- When unused, calculations will be performed on the entire dataset

**ORDER BY**
- Sorts rows within window frames by column or expressions
- When unused, window frame rows are processed in a random order

**FRAME CLAUSE**
- Used to include/exclude rows within each window frame when performing calculations
- Default frame clause is RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

* Window functions are not allowed in HAVING
* We can only use the outputs from the GROUP BY query in the HAVING clause.



1. Use a Random Function in the `WHERE` Clause

In many SQL dialects, you can use a random number function directly in the `WHERE` clause to filter rows probabilistically. This method can work well if you want an approximate sample size or random selection based on a specific probability.

PostgreSQL

In PostgreSQL, you can use the `RANDOM()` function directly in the `WHERE` clause:

```sql
SELECT *
FROM your_table
WHERE RANDOM() < 0.1;
```

- Explanation: `RANDOM()` generates a random floating-point number between 0 and 1 for each row. The condition `RANDOM() < 0.1` will approximately select 10% of the rows, providing a random sample without sorting the entire dataset.

MySQL

In MySQL, the equivalent function is `RAND()`:

```sql
SELECT *
FROM your_table
WHERE RAND() < 0.1;
```

- Explanation: Similar to PostgreSQL, `RAND()` generates a random number between 0 and 1 for each row. The condition `RAND() < 0.1` will approximately sample 10% of the rows.

* PERCENT_RANK() vs CUME_DIST() vs NTILE()

* ORDER BY NULLS FIRST NULLS LAST

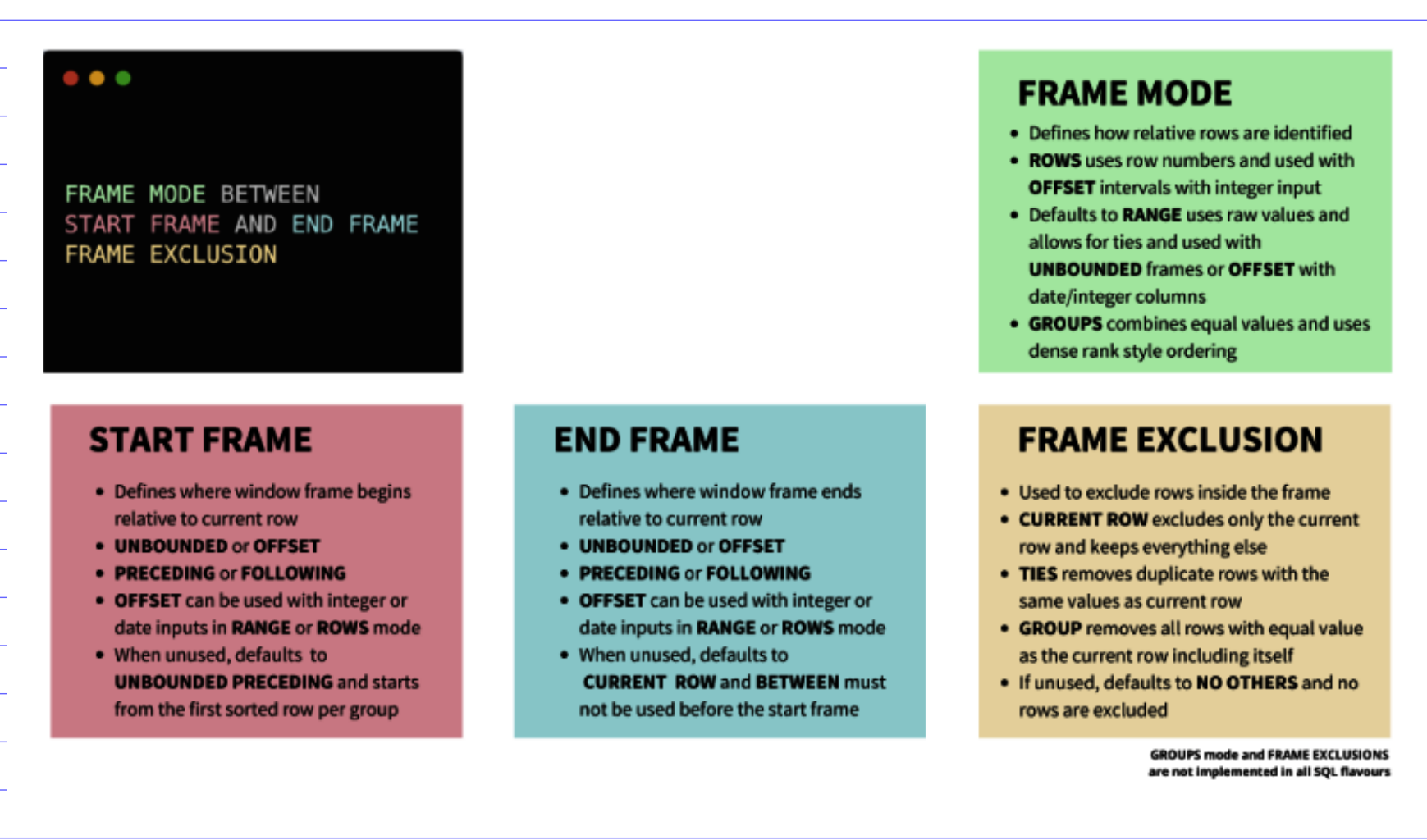
→ WINDOW Clause Simplification: if we find ourselves repeating a certain window multiple times in the same query, we can use the WINDOW clause after the final FROM section of the SQL query & refer to a simple alias in the window function call within the select expressions in the body of the SQL query.

```
Select
   Sum(<col>) OVER w
FROM <table>
WINDOW
   w AS ( ~~~~ ),
   w2 AS ( ~~~~ );
```

→ SUM(<col>) OVER (ORDER BY ____ ____)

↳ cumulative sum

→ Default FRAME: RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW



```
FRAME MODE BETWEEN
START FRAME AND END FRAME
FRAME EXCLUSION
```

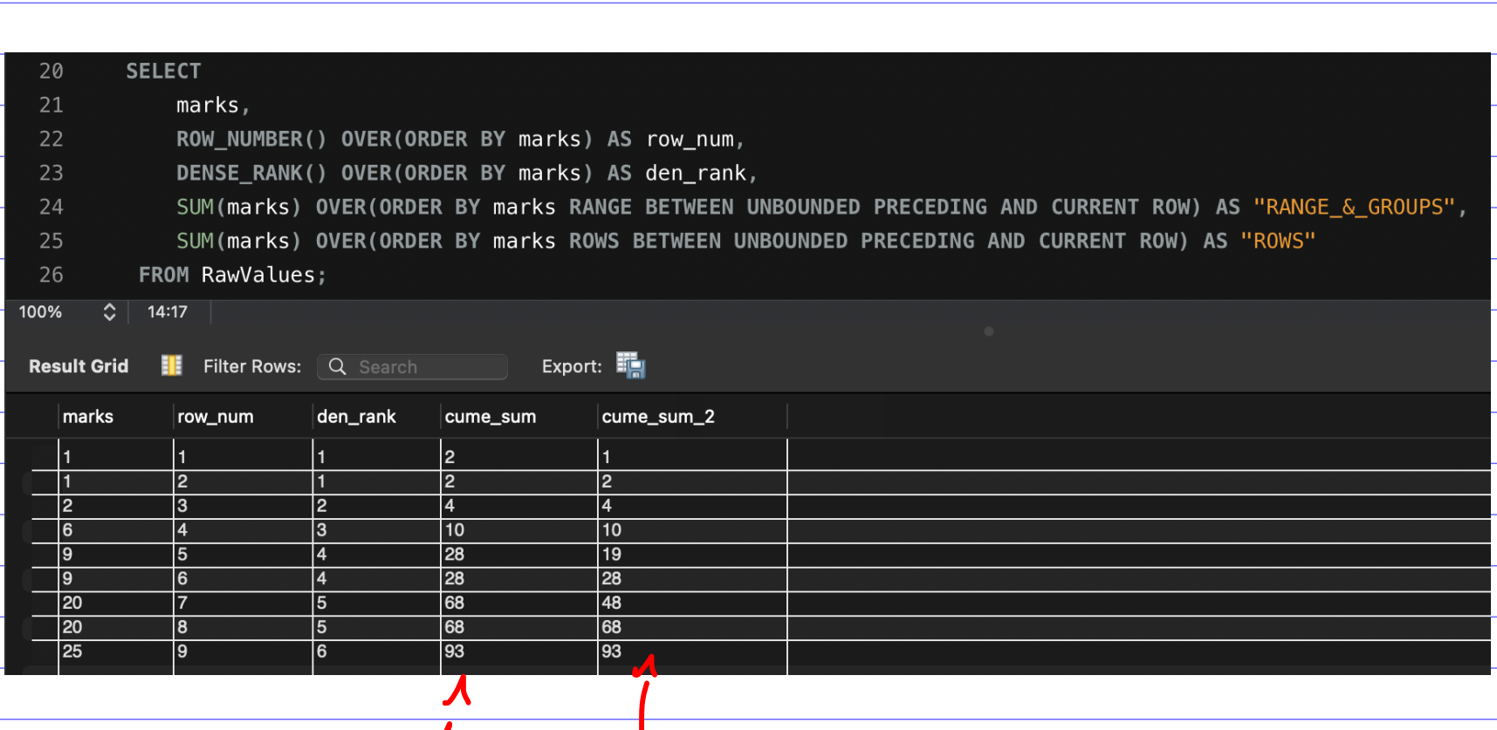
**FRAME MODE**
- Defines how relative rows are identified
- **ROWS** uses row numbers and used with **OFFSET** intervals with integer input
- Defaults to **RANGE** uses raw values and allows for ties and used with **UNBOUNDED** frames or **OFFSET** with date/integer columns
- **GROUPS** combines equal values and uses dense rank style ordering

**START FRAME**
- Defines where window frame begins relative to current row
- **UNBOUNDED** or **OFFSET**
- **PRECEDING** or **FOLLOWING**
- **OFFSET** can be used with integer or date inputs in **RANGE** or **ROWS** mode
- When unused, defaults to **UNBOUNDED PRECEDING** and starts from the first sorted row per group

**END FRAME**
- Defines where window frame ends relative to current row
- **UNBOUNDED** or **OFFSET**
- **PRECEDING** or **FOLLOWING**
- **OFFSET** can be used with integer or date inputs in **RANGE** or **ROWS** mode
- When unused, defaults to **CURRENT ROW** and **BETWEEN** must not be used before the start frame

**FRAME EXCLUSION**
- Used to exclude rows inside the frame
- **CURRENT ROW** excludes only the current row and keeps everything else
- **TIES** removes duplicate rows with the same values as current row
- **GROUP** removes all rows with equal value as the current row including itself
- If unused, defaults to **NO OTHERS** and no rows are excluded

GROUPS mode and FRAME EXCLUSIONS are not implemented in all SQL flavours

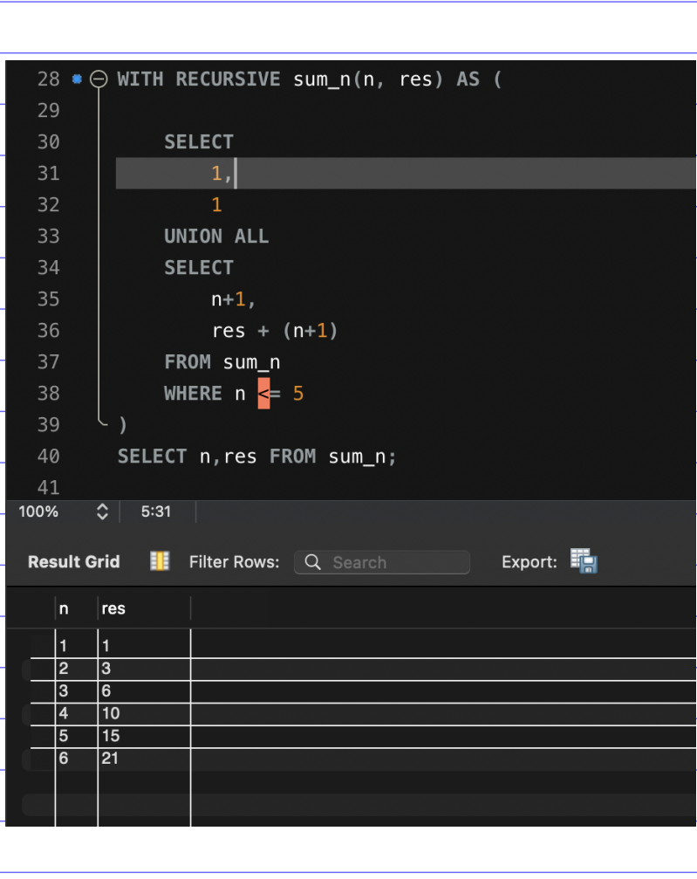


```
20   SELECT
21       marks,
22       ROW_NUMBER() OVER(ORDER BY marks) AS row_num,
23       DENSE_RANK() OVER(ORDER BY marks) AS den_rank,
24       SUM(marks) OVER(ORDER BY marks RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS "RANGE_&_GROUPS",
25       SUM(marks) OVER(ORDER BY marks ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS "ROWS"
26   FROM RawValues;
```

| marks | row_num | den_rank | cume_sum | cume_sum_2 |
|---|---|---|---|---|
| 1 | 1 | 1 | 2 | 1 |
| 1 | 2 | 1 | 2 | 2 |
| 2 | 3 | 2 | 4 | 4 |
| 6 | 4 | 3 | 10 | 10 |
| 9 | 5 | 4 | 28 | 19 |
| 9 | 6 | 4 | 28 | 28 |
| 20 | 7 | 5 | 68 | 48 |
| 20 | 8 | 5 | 68 | 68 |
| 25 | 9 | 6 | 93 | 93 |

actual output using default behaviour

expected output



```
28   WITH RECURSIVE sum_n(n, res) AS (
29
30       SELECT
31           1,
32           1
33       UNION ALL
34       SELECT
35           n+1,
36           res + (n+1)
37       FROM sum_n
38       WHERE n <= 5
39   )
40   SELECT n,res FROM sum_n;
41
```

| n | res |
|---|---|
| 1 | 1 |
| 2 | 3 |
| 3 | 6 |
| 4 | 10 |
| 5 | 15 |
| 6 | 21 |

Experimentation with Recursive table
sum 'n' natural numbers

PRACTICE MORE

→ FRAME MODES

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 1 | 2 | 6 | 9 | 9 | 20 | 20 | 25 |
| ROWS | 4 | 3 | 2 | 1 | | 1 | 2 | 3 | 4 |
| | 1 | 1 | 2 | 6 | 9 | 9 | 20 | 20 | 25 |
| RANGE | 8 | 8 | 7 | 3 | | | 11 | 11 | 16 |
| | 1 | 1 | 2 | 6 | 9 | 9 | 20 | 20 | 25 |
| GROUPS | 3 | 3 | 2 | 1 | | | 1 | 1 | 2 |

UNBOUNDED, PRECEDING, FOLLOWING, CURRENT ROW

FRAME Exclusion
- CURRENT ROW
- TIES
- GROUP
- NO OTHERS (DEFAULT)

PRACTICAL USE CASES??