→ Temporary tables & CTE

TEMP TABLE IF EXISTS [table_name];
CREATE TEMP TABLE [table_name] AS (

```
WITH [cte_name] (col_1, col_2, col_3) AS (
    VALUES
        ('', '', '',),
        ('', '', '',),
        ('', '', '',)
)
```

)

## Full Join

## Cross Join

→ FROM t1 CROSS JOIN t2 / FROM t1, t2

- ON / USING clause cannot be used
- Cartesian product
- Rows from the first table are combined with every row from the second table, producing a result set that includes all possible combinations of rows from both tables.
- Used when you need to create all possible combinations between two datasets.

## Left Semi Join → Subquery

- Returns only rows from the first table (left table) that have matching rows in the second table (right table).
- Ensures that each row from the first table appears at most once in the result, even if there are multiple matches in the second table.
- Commonly used in subqueries or when checking for the existence of related data.
- Syntax varies slightly between database systems, but the concept remains the same.

→ Target table made on a filter

Syntax

```
SELECT
    *
FROM [base_table]
WHERE EXISTS (
    SELECT
        1
    FROM [target_table]
    WHERE [join_key] = [base_table].[join_key]
);
```

Syntax 2

```
SELECT
    *
FROM [base_table]
WHERE [join_key] IN (
    SELECT
        [join_key]
    FROM [target_table]
    WHERE [join_key] = [base_table].[join_key]
);
```

→ Semi-join

→ IN or EXISTS

→ Subquery returns only values from the first table (base_table) that have matching values in the second table (target_table).

→ Use EXISTS when you need to check for the presence of related data without actually joining the tables.

→ Use IN when you need to compare a single column against a list of values.

→ Both EXISTS and IN can be used with correlated subqueries, where the subquery depends on the outer query.

→ Performance depends on the database system and the specific query. Modern database systems often optimize both EXISTS and IN similarly.

→ EXISTS is generally more efficient when dealing with large datasets or when the subquery returns many rows.

→ IN can be more readable for small, static lists of values.

## Anti Join

- Returns rows from the first table (left table) that do not have matching rows in the second table (right table).
- Ensures that each row from the first table appears at most once in the result, even if there are multiple matches in the second table.
- Commonly used in subqueries or when checking for the absence of related data.
- Syntax varies slightly between database systems, but the concept remains the same.

Syntax

```
SELECT
    *
FROM [base_table]
WHERE NOT EXISTS (
    SELECT
        1
    FROM [target_table]
    WHERE [join_key] = [base_table].[join_key]
);
```

Syntax 2

```
SELECT
    *
FROM [base_table]
WHERE [join_key] NOT IN (
    SELECT
        [join_key]
    FROM [target_table]
    WHERE [join_key] IS NOT NULL
);
```

OR

NOT EXISTS → MORE EFFICIENT

```
SELECT
    *
FROM [base_table] AS t1
LEFT JOIN [target_table] AS t2
    ON t1.[join_key] = t2.[join_key]
WHERE t2.[join_key] IS NULL;
```

→ If we are joining tables which have NULLS in the JOINING COLUMN THE RESULTS DON'T AS EXPECTED. NULL = NULL

Symmetric difference

```
SELECT
    1 + NULL AS col1,
    2 + NULL AS col2,
    3 + NULL AS col3,
    4 + NULL AS col4,    -- this is our unknown value
    5 + NULL AS col5,    -- this is our unknown value
    6 + NULL AS col6,
    7 + NULL AS col7;
```

| col1 | col2 | col3 | col4 | col5 | col6 | col7 |
|---|---|---|---|---|---|---|
| NULL | NULL | NULL | NULL | NULL | NULL | NULL |

Table 1: Results of the query

Technically things already work that perfectly as because, in our always-plus-one query, COALESCE(col, 0) + 1 and NULL + 1 both evaluate to NULL. So, the unknown value propagates through the calculation, which is actually the correct behavior in most cases.

## Union

Syntax: SELECT * FROM table1 UNION SELECT * FROM table2;

## Union All

## Intersect

## Except

- UNION, INTERSECT, EXCEPT: Combine rows from multiple tables.
- JOIN: Combines columns from multiple tables based on a condition.

→ UPDATE [table]
SET
    [col1] = _____,
    [col2] = _____,
WHERE _____;
RETURNING *;

→ Shows all updated rows as query output

→ Monitoring with row-level versions using the xmin system column

- xmin stands for transaction ID (XID) of the transaction that created the row version.
- xmin is a system column in PostgreSQL that stores the transaction ID (XID) of the transaction that last modified the row.
- xmin is used for multi-version concurrency control (MVCC) to track which transaction created each row version.
- xmin is a 32-bit integer that wraps around after approximately 4 billion transactions.
- xmin is used by VACUUM to determine which row versions can be safely removed.
- xmin is used by SERIALIZABLE transactions to detect serialization anomalies.
- xmin is used by logical replication to track which changes have been applied.
- xmin is used by pg_stat_activity to show the oldest transaction that is still active.
- xmin is used by pg_stat_all_tables to track table activity.
- xmin is used by pg_stat_all_indexes to track index activity.
- xmin is used by pg_stat_bgwriter to track background writer activity.
- xmin is used by pg_stat_checkpointer to track checkpoint activity.
- xmin is used by pg_stat_wal to track WAL activity.
- xmin is used by pg_stat_wal_receiver to track WAL receiver activity.
- xmin is used by pg_stat_wal_sender to track WAL sender activity.
- xmin is used by pg_stat_replication to track replication activity.
- xmin is used by pg_stat_replication_slots to track replication slot activity.
- xmin is used by pg_stat_subscription to track subscription activity.
- xmin is used by pg_stat_subscription_stats to track subscription statistics.
- xmin is used by pg_stat_ssl to track SSL activity.
- xmin is used by pg_stat_gssapi to track GSSAPI activity.
- xmin is used by pg_stat_archiver to track archiver activity.
- xmin is used by pg_stat_bgwriter to track background writer activity.
- xmin is used by pg_stat_checkpointer to track checkpoint activity.
- xmin is used by pg_stat_wal to track WAL activity.
- xmin is used by pg_stat_wal_receiver to track WAL receiver activity.
- xmin is used by pg_stat_wal_sender to track WAL sender activity.
- xmin is used by pg_stat_replication to track replication activity.
- xmin is used by pg_stat_replication_slots to track replication slot activity.
- xmin is used by pg_stat_subscription to track subscription activity.
- xmin is used by pg_stat_subscription_stats to track subscription statistics.
- xmin is used by pg_stat_ssl to track SSL activity.
- xmin is used by pg_stat_gssapi to track GSSAPI activity.
- xmin is used by pg_stat_archiver to track archiver activity.

→ Updating with row-level versions using the xmin system column

- xmin stands for transaction ID (XID) of the transaction that created the row version.
- xmin is a system column in PostgreSQL that stores the transaction ID (XID) of the transaction that last modified the row.
- xmin is used for multi-version concurrency control (MVCC) to track which transaction created each row version.
- xmin is a 32-bit integer that wraps around after approximately 4 billion transactions.
- xmin is used by VACUUM to determine which row versions can be safely removed.
- xmin is used by SERIALIZABLE transactions to detect serialization anomalies.
- xmin is used by logical replication to track which changes have been applied.
- xmin is used by pg_stat_activity to show the oldest transaction that is still active.
- xmin is used by pg_stat_all_tables to track table activity.
- xmin is used by pg_stat_all_indexes to track index