Essential SQL Concepts

Every Data Engineer
Should Master

UNION VS UNION ALL

UNION:

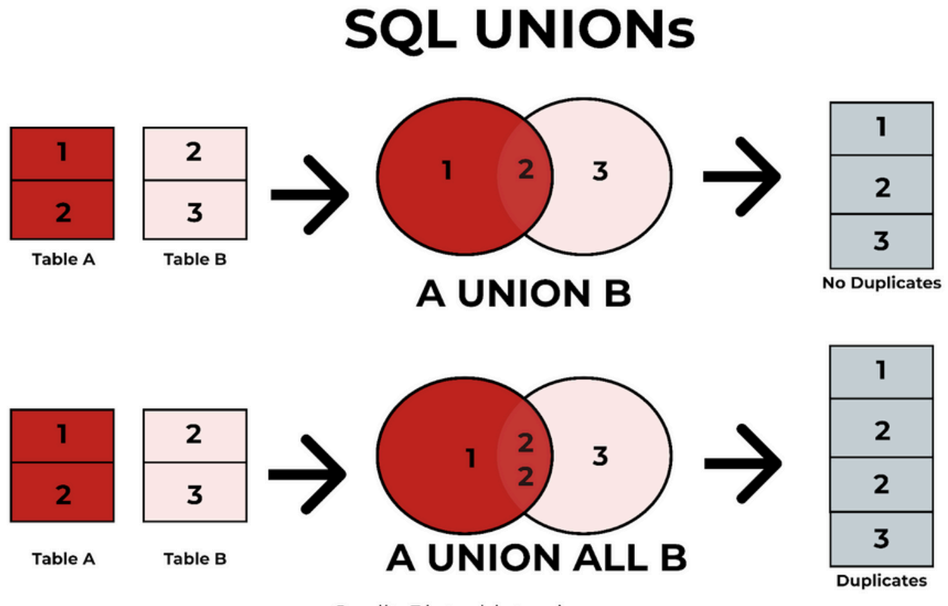
SELECT COLUMN1, COLUMN2 FROM TABLE1 UNION SELECT COLUMN1, COLUMN2 FROM TABLE2

COMBINES RESULTS FROM TWO OR MORE QUERIES, AUTOMATICALLY REMOVING DUPLICATE ROWS. IT PERFORMS A DISTINCT OPERATION, WHICH CAN AFFECT PERFORMANCE, ESPECIALLY ON LARGE DATASETS.

UNION ALL:

SELECT COLUMN1, COLUMN2 FROM TABLE1 UNION ALL SELECT COLUMN1, COLUMN2 FROM TABLE2

COMBINES RESULTS AND INCLUDES DUPLICATES FROM ALL QUERIES. IT IS FASTER THAN UNION BECAUSE IT DOESN'T HAVE TO CHECK FOR DUPLICATES.



CTE VS TEMP VS TABLE VARIABLE

CTE:

WITH CTENAME AS (SELECT COLUMN1, COLUMN2 FROM TABLE1)

SELECT * FROM CTENAME

A TEMPORARY RESULT SET DEFINED WITHIN A QUERY, IDEAL FOR SIMPLIFYING COMPLEX QUERIES, SUPPORTING RECURSION, AND IMPROVING READABILITY. IT EXISTS ONLY FOR THE DURATION OF THE QUERY EXECUTION.

Temporary Table

```
DROP TABLE IF EXISTS #tempManager


SELECT employee_id, MAX(manager_name) AS manager_name
INTO #tempManager
FROM Sales
GROUP BY employee_id
```

VS

Common Table Expression (“CTE”)

```
WITH cteManager AS (
  SELECT employee_id, MAX(manager_name) AS manager_name
  FROM Sales
  GROUP BY employee_id
)
```

CTE VS TEMP VS TABLE VARIABLE

TEMPORARY TABLE:

CREATE TABLE #TEMPTABLE (COLUMN1 DATATYPE, COLUMN2 DATATYPE)

INSERT INTO #TEMPTABLE SELECT COLUMN1, COLUMN2 FROM TABLE1

STORED IN THE TEMPDB DATABASE AND ACTS AS A PHYSICAL TABLE. TEMPORARY TABLES SUPPORT INDEXING, MAKING THEM SUITABLE FOR LARGE DATASETS OR OPERATIONS REQUIRING MULTIPLE MANIPULATIONS. THEY PERSIST FOR THE DURATION OF THE SESSION OR BATCH EXECUTION.

Temporary Table

```
DROP TABLE IF EXISTS #tempManager


SELECT employee_id, MAX(manager_name) AS manager_name
INTO #tempManager
FROM Sales
GROUP BY employee_id
```

VS

Common Table Expression (“CTE”)

```
WITH cteManager AS (
  SELECT employee_id, MAX(manager_name) AS manager_name
  FROM Sales
  GROUP BY employee_id
)
```

CTE VS TEMP VS TABLE VARIABLE

TABLE VARIABLE: ***DECLARE @TABLEVARIABLE TABLE (COLUMN1 DATATYPE, COLUMN2 DATATYPE) INSERT INTO @TABLEVARIABLE SELECT COLUMN1, COLUMN2 FROM TABLE1***

IN-MEMORY STRUCTURES USED FOR LIGHTWEIGHT OPERATIONS ON SMALLER DATASETS. TABLE VARIABLES ARE LESS RESOURCE-INTENSIVE BUT HAVE LIMITED FUNCTIONALITY COMPARED TO TEMPORARY TABLES (E.G., NO SUPPORT FOR INDEXING IN MOST CASES).

Temporary Table

```
DROP TABLE IF EXISTS #tempManager


SELECT employee_id, MAX(manager_name) AS manager_name
INTO #tempManager
FROM Sales
GROUP BY employee_id
```

VS

Common Table Expression (“CTE”)

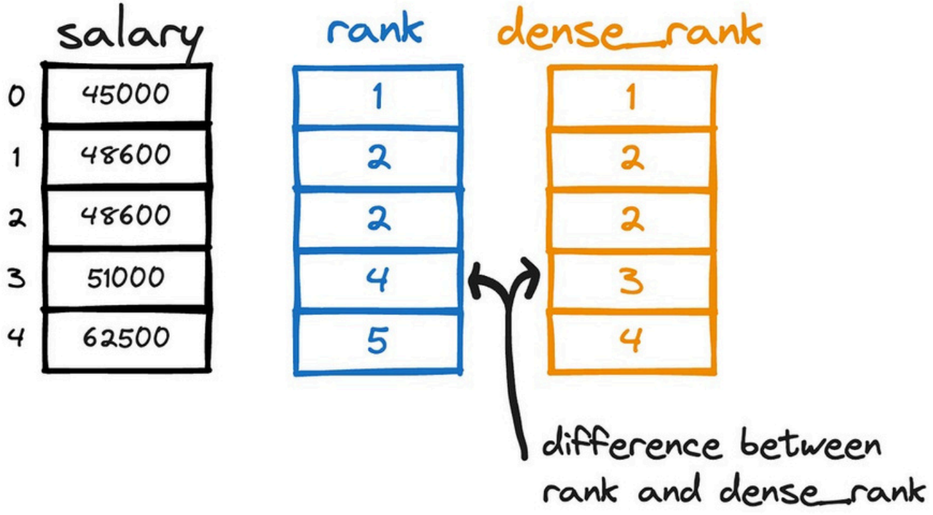
```
WITH cteManager AS (
  SELECT employee_id, MAX(manager_name) AS manager_name
  FROM Sales
  GROUP BY employee_id
)
```

RANK VS ROW_NUMBER VS DENSE_RANK

RANK:

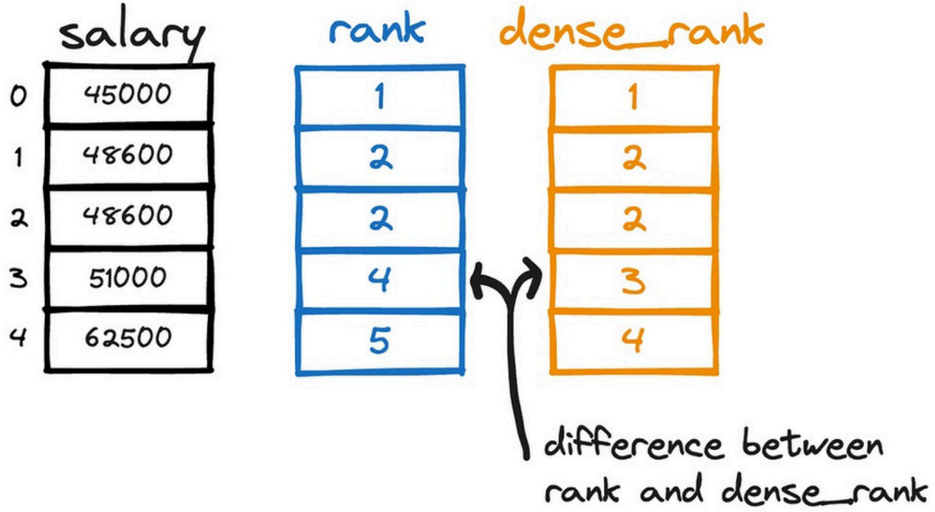
SELECT COLUMN1, RANK() OVER (ORDER BY COLUMN2) AS RANK FROM TABLE1

ASSIGNS A RANK TO EACH ROW BASED ON A SPECIFIED ORDER. IF TWO ROWS HAVE THE SAME RANK, THE NEXT RANK IS SKIPPED. FOR EXAMPLE, IF TWO ROWS ARE RANKED 2, THE NEXT RANK WILL BE 4. USEFUL FOR SCENARIOS WHERE YOU NEED TO SHOW RELATIVE POSITIONING BUT MAINTAIN GAPS FOR TIES.



RANK VS ROW_NUMBER VS DENSE_RANK

ROW_NUMBER: ***SELECT COLUMN1, ROW_NUMBER() OVER (ORDER BY COLUMN2) AS ROWNUM FROM TABLE1*** ASSIGNS A UNIQUE NUMBER TO EACH ROW, REGARDLESS OF TIES. THIS IS PARTICULARLY USEFUL FOR PAGINATION OR UNIQUELY IDENTIFYING ROWS IN A DATASET.

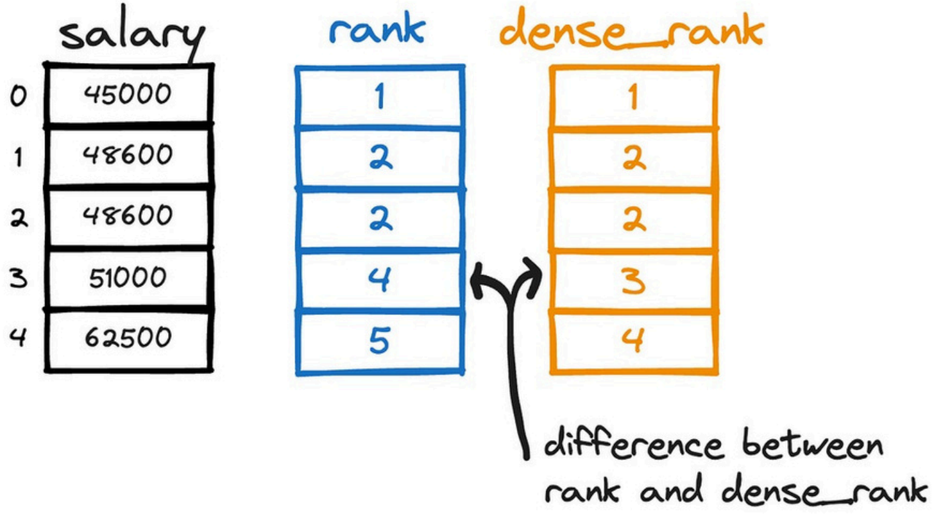


RANK VS ROW_NUMBER VS DENSE_RANK

DENSE_RANK:

SELECT COLUMN1, DENSE_RANK() OVER (ORDER BY COLUMN2) AS DENSERANK FROM TABLE1

SIMILAR TO RANK BUT WITHOUT GAPS IN NUMBERING FOR TIES. FOR EXAMPLE, IF TWO ROWS ARE RANKED 2, THE NEXT RANK WILL BE 3. IDEAL FOR LEADERBOARD SYSTEMS WHERE CONSECUTIVE NUMBERING IS PREFERRED.

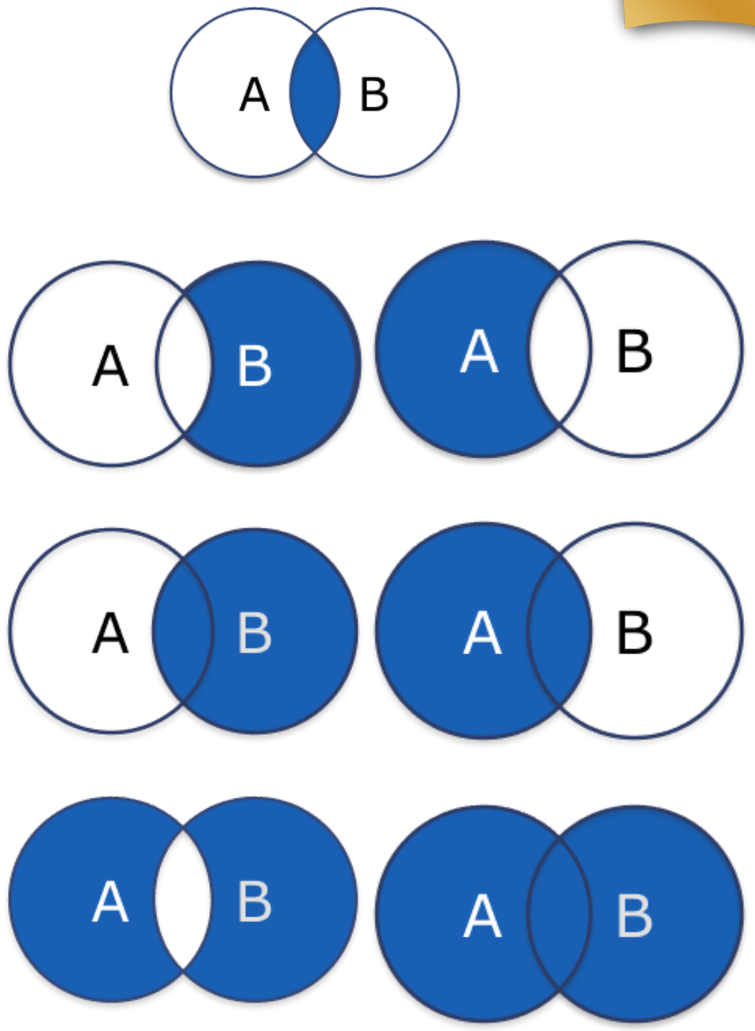


JOINS

INNER JOIN:

SELECT * FROM TABLE1 INNER JOIN TABLE2 ON TABLE1.ID = TABLE2.ID

RETURNS ONLY THE ROWS WHERE THERE IS A MATCH BETWEEN THE TWO TABLES BASED ON THE JOIN CONDITION. IT'S THE DEFAULT AND MOST COMMONLY USED JOIN TYPE WHEN YOU NEED TO COMBINE DATA WITH EXACT RELATIONSHIPS.

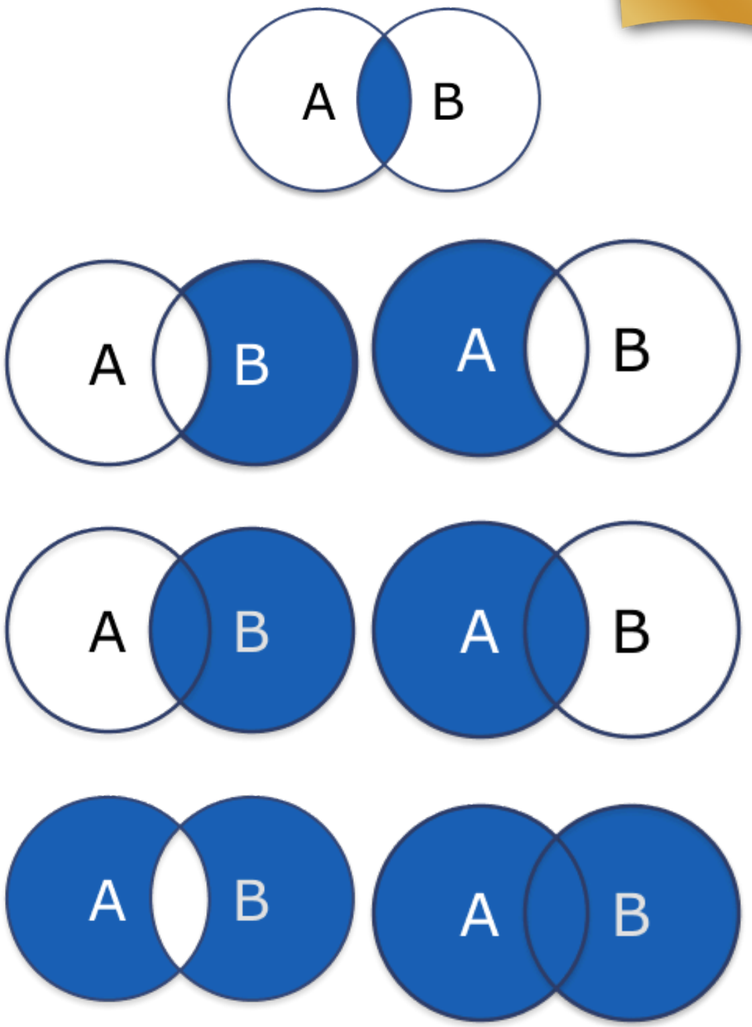


JOINS

OUTER JOIN:

- **LEFT JOIN:** INCLUDES ALL ROWS FROM THE LEFT TABLE AND MATCHING ROWS FROM THE RIGHT TABLE.
- **RIGHT JOIN:** INCLUDES ALL ROWS FROM THE RIGHT TABLE AND MATCHING ROWS FROM THE LEFT TABLE.
- **FULL JOIN:** INCLUDES ALL ROWS FROM BOTH TABLES, WITH NULLS WHERE THERE'S NO MATCH.

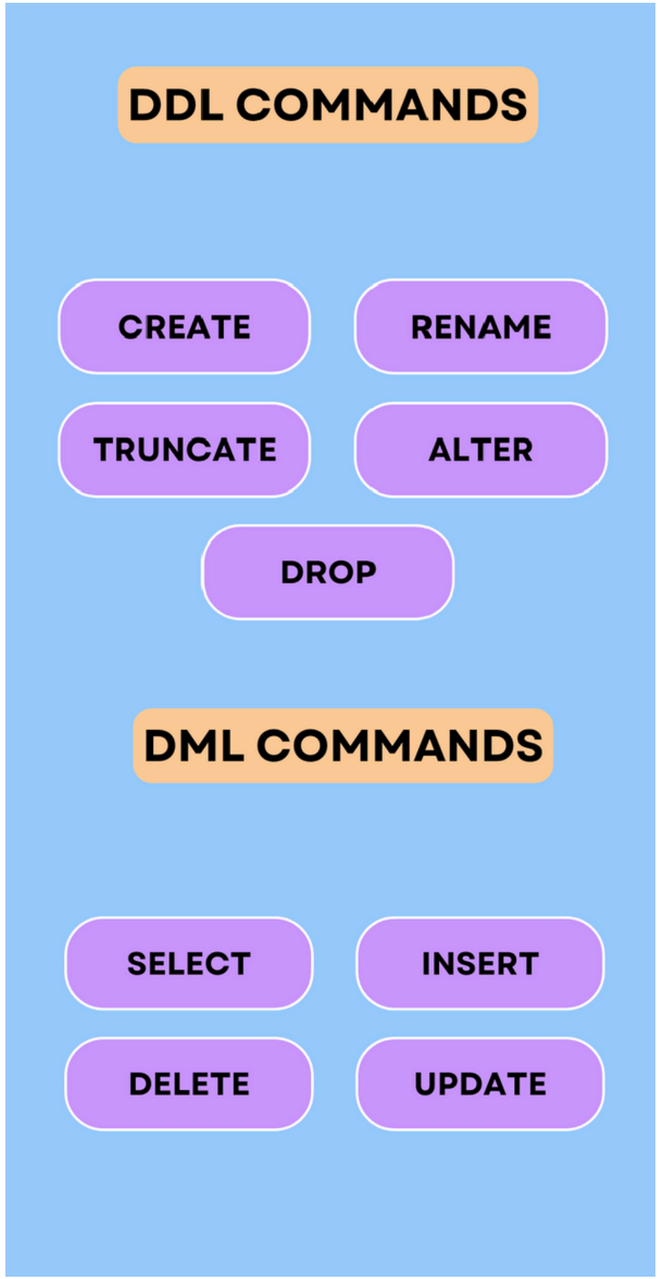
THESE JOINS ARE USED WHEN YOU NEED TO RETAIN UNMATCHED ROWS FROM ONE OR BOTH TABLES.



DDL VS DML

DDL (DATA DEFINITION LANGUAGE): *CREATE TABLE TABLE_NAME (COLUMN1 DATATYPE, COLUMN2 DATATYPE)*

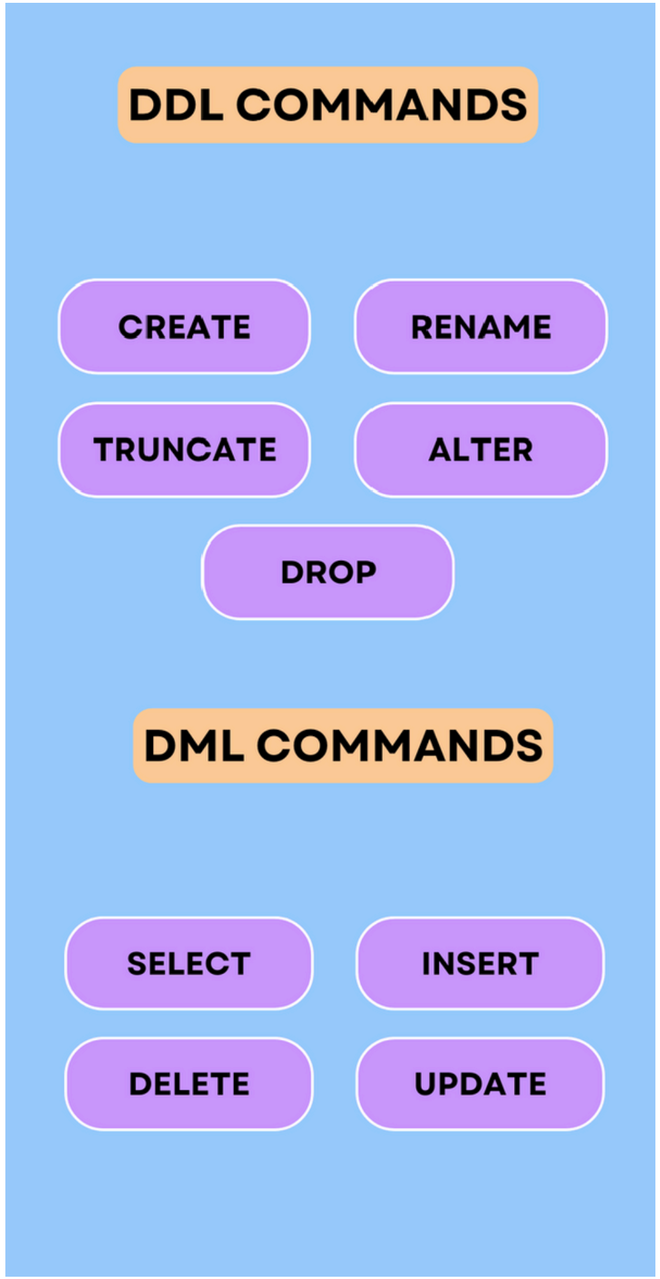
DEFINES, ALTERS, OR DELETES DATABASE STRUCTURES LIKE TABLES, SCHEMAS, AND INDEXES. EXAMPLES INCLUDE CREATE, ALTER, AND DROP. IT'S ESSENTIAL FOR DATABASE DESIGN AND STRUCTURE DEFINITION.



DDL VS DML

DML (DATA MANIPULATION LANGUAGE): ***INSERT INTO TABLE_NAME VALUES (VALUE1, VALUE2)***

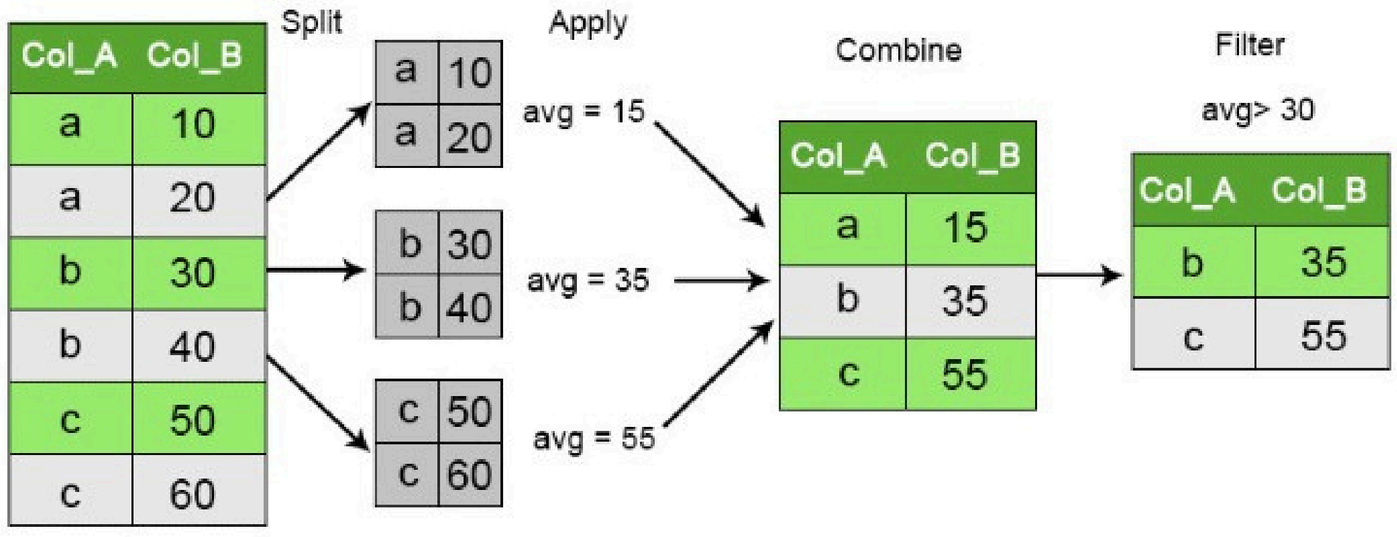
USED FOR INTERACTING WITH THE DATA STORED IN A DATABASE. EXAMPLES INCLUDE SELECT, INSERT, UPDATE, AND DELETE. DML QUERIES ARE FUNDAMENTAL FOR WORKING WITH AND ANALYZING DATA.



WHERE VS HAVING

WHERE: *SELECT * FROM TABLE1 WHERE COLUMN1 = 'VALUE'*

FILTERS ROWS BEFORE AGGREGATION OR GROUPING. IT APPLIES CONDITIONS TO INDIVIDUAL ROWS IN A DATASET AND IS USED FOR FILTERING AT THE RAW DATA LEVEL.

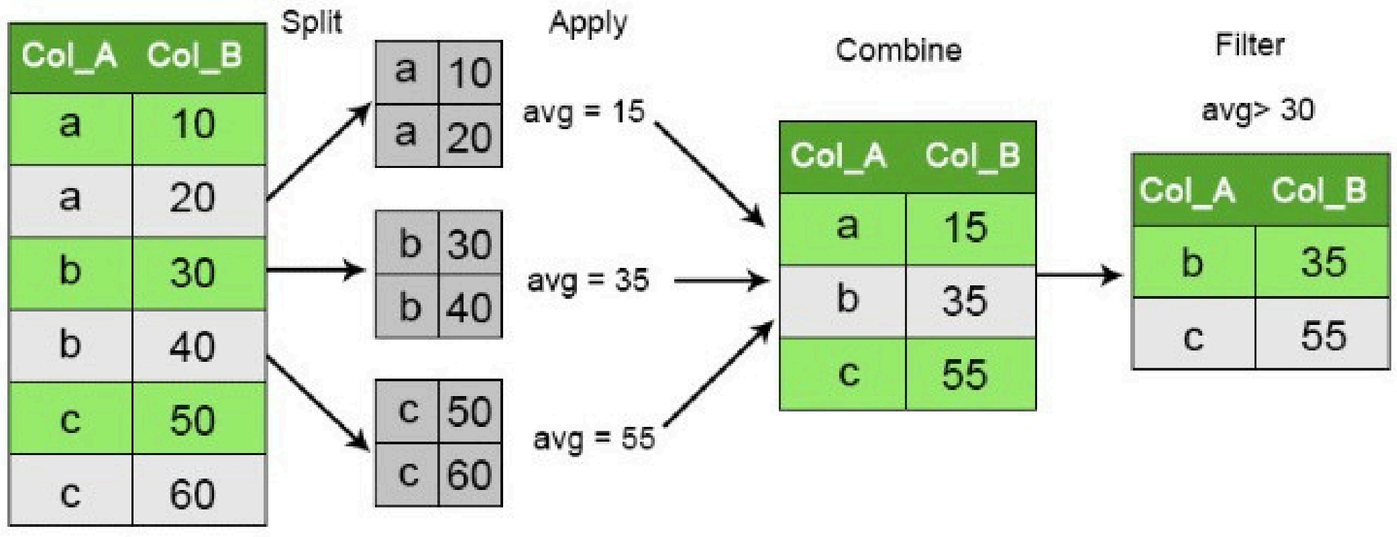


WHERE VS HAVING

HAVING:

SELECT COLUMN1, COUNT(*) FROM TABLE1 GROUP BY COLUMN1 HAVING COUNT(*) > 5

FILTERS RESULTS AFTER GROUPING OR AGGREGATION. IT'S APPLIED TO AGGREGATE FUNCTIONS LIKE COUNT, SUM, OR AVG, ENSURING CONDITIONS ARE CHECKED AT A SUMMARY LEVEL.

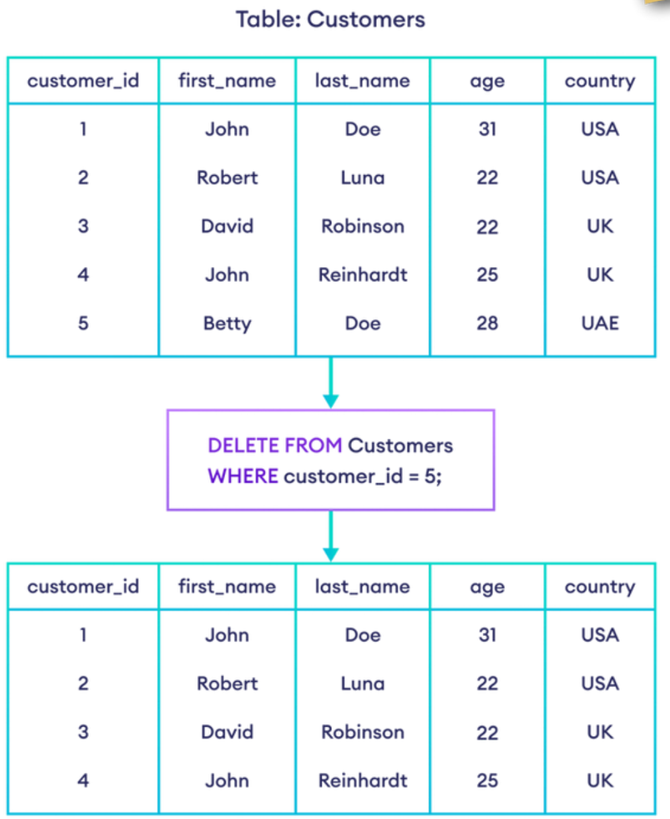


TRUNCATE VS DELETE

TRUNCATE:

TRUNCATE TABLE TABLE_NAME

REMOVES ALL ROWS FROM A TABLE QUICKLY AND EFFICIENTLY BY DEALLOCATING DATA PAGES. IT RESETS IDENTITY COLUMNS BUT DOESN'T LOG INDIVIDUAL ROW DELETIONS. SUITABLE FOR BULK DATA REMOVAL WHEN LOGGING ISN'T NECESSARY.

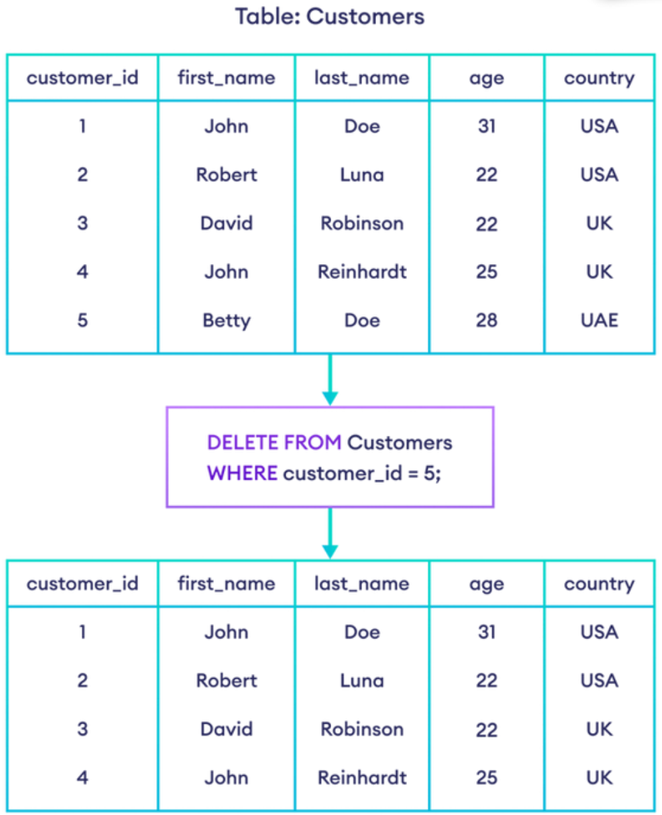


TRUNCATE VS DELETE

DELETE:

DELETE FROM TABLE_NAME WHERE COLUMN1 = 'VALUE'

REMOVES SPECIFIC ROWS FROM A TABLE. IT LOGS EACH DELETION AND RETAINS IDENTITY COLUMN VALUES. USE DELETE WHEN YOU NEED TO SELECTIVELY REMOVE ROWS OR WHEN LOGGING IS REQUIRED FOR AUDIT PURPOSES.

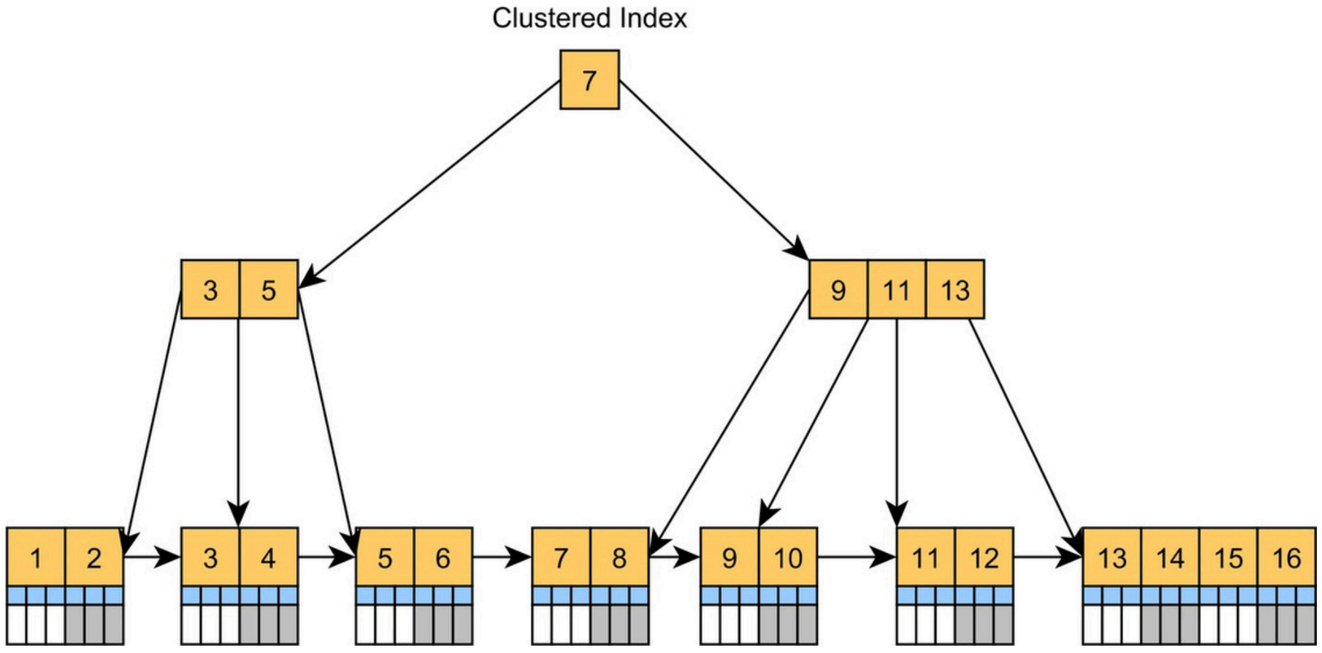


Clustered Index VS Non-Clustered Index

CLUSTERED INDEX:

CREATE CLUSTERED INDEX IDX_NAME ON TABLE_NAME(COLUMN1)

SORTS AND STORES THE DATA ROWS IN THE TABLE BASED ON THE KEY VALUES. EACH TABLE CAN HAVE ONLY ONE CLUSTERED INDEX. BEST USED FOR COLUMNS FREQUENTLY SEARCHED OR USED FOR SORTING, LIKE PRIMARY KEYS.

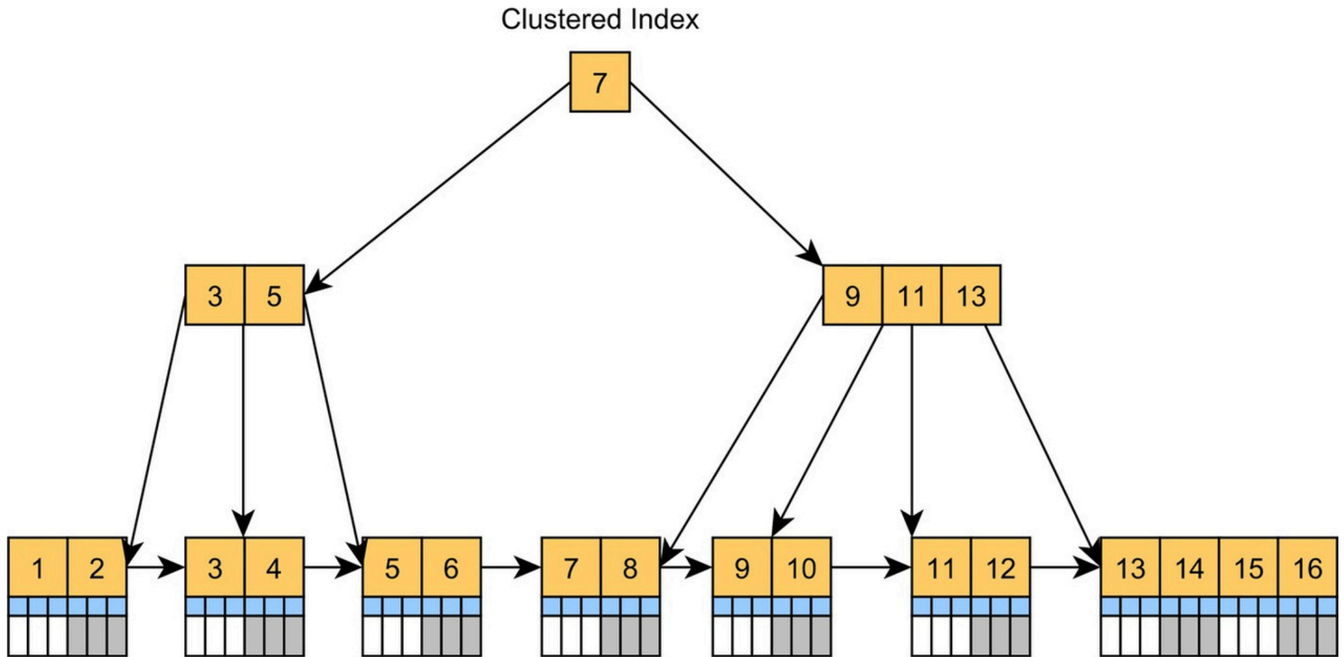


Clustered Index VS Non-Clustered Index

NON-CLUSTERED INDEX:

CREATE NONCLUSTERED INDEX IDX_NAME ON TABLE_NAME(COLUMN1)

STORES POINTERS TO THE DATA ROWS INSTEAD OF SORTING THE DATA ITSELF. MULTIPLE NON-CLUSTERED INDEXES CAN EXIST ON A SINGLE TABLE, MAKING THEM IDEAL FOR SPEEDING UP SPECIFIC QUERIES.



COALESCE VS ISNULL

COALESCE:

SELECT COALESCE(COLUMN1, COLUMN2, 'DEFAULT') FROM TABLE1

RETURNS THE FIRST NON-NULL VALUE FROM A LIST OF INPUTS. IT'S HIGHLY FLEXIBLE AND CAN HANDLE MULTIPLE COLUMNS OR VALUES, MAKING IT A ROBUST OPTION FOR HANDLING NULLS ACROSS DATASETS.

Show this column with COALESCE()

employee_id	employee_name	points_original	points_new	points_final
1	Dexter DeShawn	75	*NULL*	75
2	Evelyn Parker	40	60	60
3	Jackie Welles	95	*NULL*	95
4	Judy Alvarez	65	*NULL*	65
5	Meredith Stout	60	80	80
6	Misty Olszewski	30	*NULL*	30
7	Viktor Vektor	80	90	90

COALESCE VS ISNULL

ISNULL:

SELECT ISNULL(COLUMN1, 'DEFAULT') FROM TABLE1

REPLACES NULL VALUES WITH A SPECIFIED DEFAULT VALUE. IT'S SIMPLER BUT LIMITED TO HANDLING ONLY ONE COLUMN OR VALUE AT A TIME.

Show this column with COALESCE()

employee_id	employee_name	points_original	points_new	points_final
1	Dexter DeShawn	75	*NULL*	75
2	Evelyn Parker	40	60	60
3	Jackie Welles	95	*NULL*	95
4	Judy Alvarez	65	*NULL*	65
5	Meredith Stout	60	80	80
6	Misty Olszewski	30	*NULL*	30
7	Viktor Vektor	80	90	90

EXISTS VS IN

EXISTS:

SELECT COLUMN1 FROM TABLE1 WHERE EXISTS (SELECT 1 FROM TABLE2 WHERE TABLE2.COLUMN1 = TABLE1.COLUMN1)

CHECKS FOR THE EXISTENCE OF ROWS IN A SUBQUERY AND STOPS AT THE FIRST MATCH. IT'S HIGHLY EFFICIENT FOR CHECKING WHETHER RELATED DATA EXISTS IN ANOTHER TABLE.

EXISTS VS IN

IN:

SELECT COLUMN1 FROM TABLE1 WHERE COLUMN1 IN (SELECT COLUMN1 FROM TABLE2)

COMPARES A VALUE TO A LIST OF VALUES OR THE RESULTS OF A SUBQUERY. IN IS STRAIGHTFORWARD BUT CAN BE LESS EFFICIENT THAN EXISTS FOR LARGE DATASETS.