SQL DIFFERENCES VISUAL GUIDE

For Data Engineers & SQL Developers

1. JOIN TYPES

INNER JOIN

Only matching rows from both tables

LEFT JOIN

All rows from left table + matches from right

RIGHT JOIN

All rows from right table + matches from left

FULL JOIN

All rows from both tables with NULLs where no match

CROSS JOIN

Cartesian product (every combination)

Key Takeaways

- ✓ INNER keeps only matches
- ✓ LEFT/RIGHT/FULL keep unmatched rows as NULL
- ✓ CROSS produces explosive row count

2. WHERE vs HAVING

	WHERE	HAVING
Applies to	✓	✗
When executed	✓	✗
Can filter by aggregates	✓	✗
Speed	✓	✗

WHERE

- Applies to: Rows (BEFORE aggregation)
- Used with: All queries
- Can filter by aggregates?: NO
- Executed: BEFORE GROUP BY
- Speed: FASTER (fewer rows grouped)

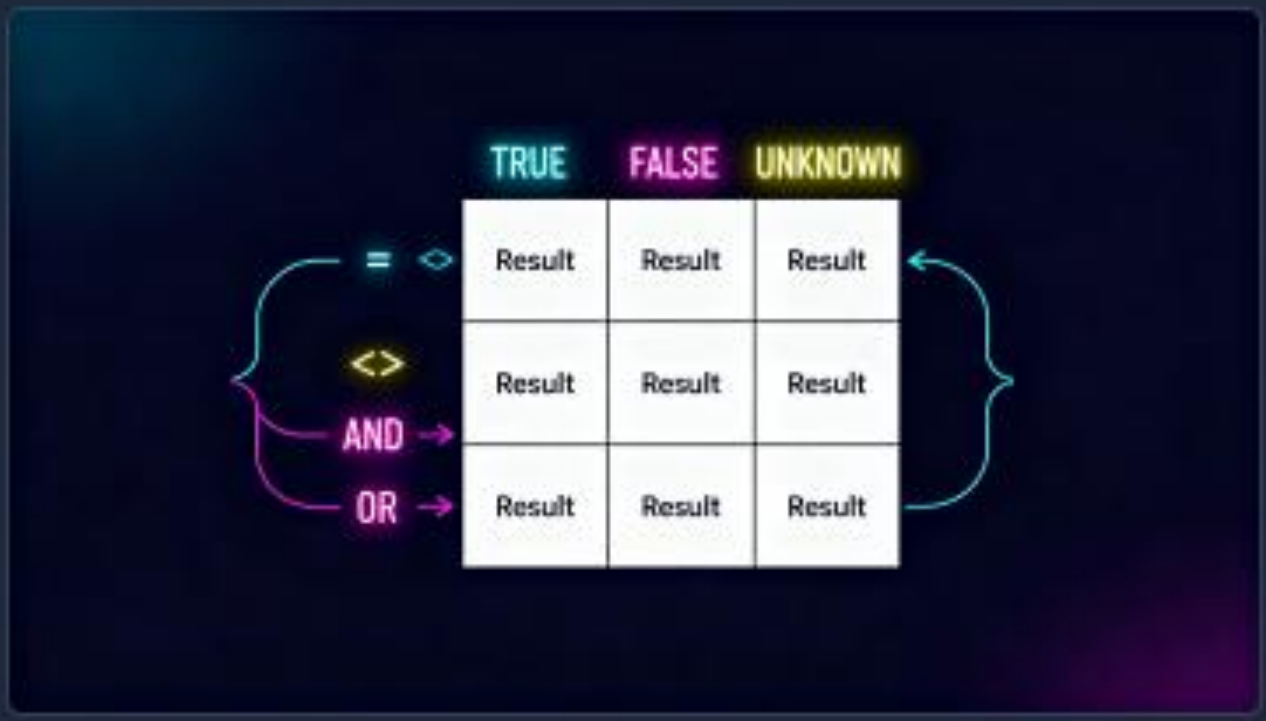
HAVING

- Applies to: Groups (AFTER aggregation)
- Used with: GROUP BY queries only
- Can filter by aggregates?: YES
- Executed: AFTER GROUP BY
- Speed: SLOWER (groups all first)

Key Takeaways

- ✓ WHERE filters data BEFORE grouping
- ✓ HAVING filters aggregated RESULTS
- ✓ WHERE placement = faster queries

3. NULL & THREE-VALUED LOGIC




```
5 = 5 → TRUE

5 = NULL → UNKNOWN

NULL = NULL → UNKNOWN

WHERE rejects: FALSE and UNKNOWN
```

⚠ Use IS NULL, never = NULL

Key Takeaways

- ✓ Three values: TRUE, FALSE, UNKNOWN
- ✓ NULL comparisons produce UNKNOWN
- ✓ WHERE keeps only TRUE rows

4. GROUP BY vs DISTINCT

DISTINCT

- Purpose: Removes duplicates only
- Aggregation: No
- Use case: Get unique values
- Example: SELECT DISTINCT *country* FROM *customers*

GROUP BY

- Purpose: Duplicates + Aggregates
- Aggregation: Yes (SUM, COUNT, AVG...)
- Use case: Get statistics per group
- Example: SELECT *country*, COUNT(*) GROUP BY *country*

Key Takeaways

- ✓ DISTINCT = deduplication
- ✓ GROUP BY = aggregation
- ✓ GROUP BY for stats, DISTINCT for unique values

5. UNION vs UNION ALL

UNION vs UNION ALL Comparison

Performance: Slower
Row Count: 500

Performance: Fasster
Row Count: 1000

UNION

- Removes duplicates: YES
- Speed: Slower
- Use: Clean merged data
- Process: Sorts and deduplicates

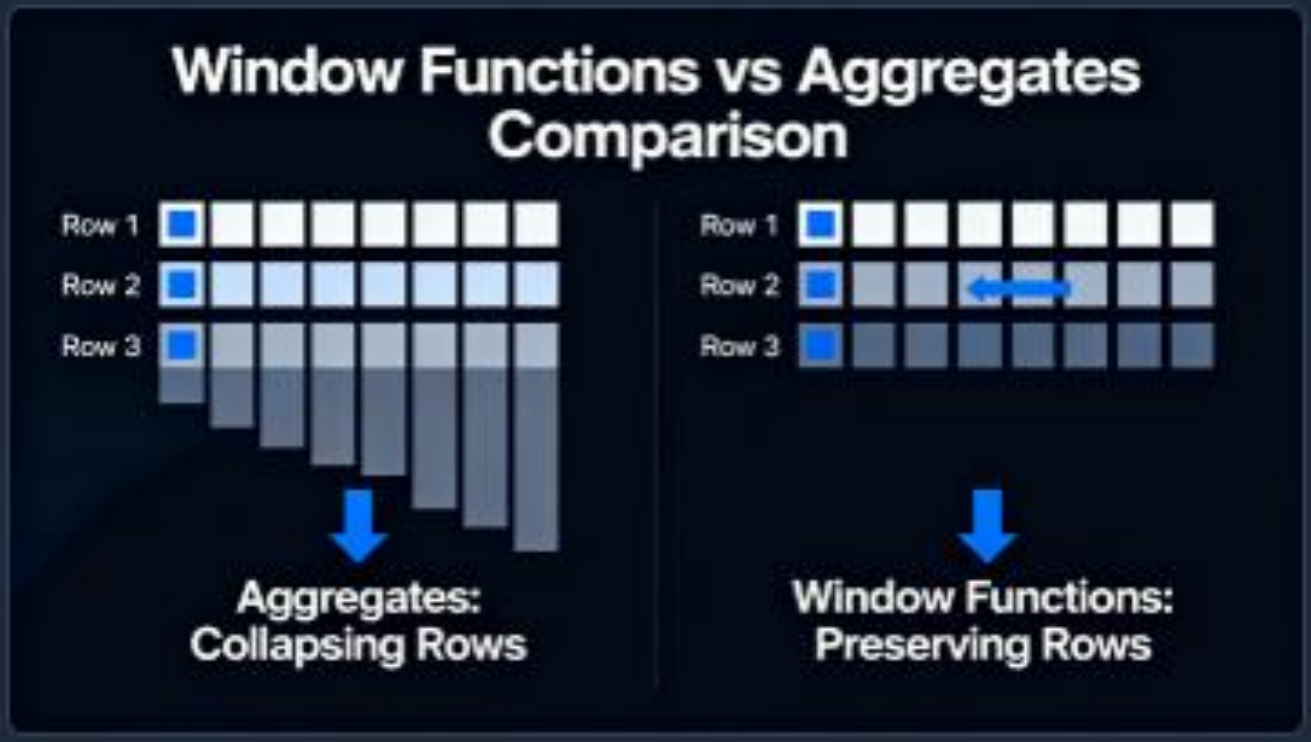
UNION ALL

- Removes duplicates: NO
- Speed: Faster
- Use: Full merged data
- Process: Simple concatenation

Key Takeaways

- ✓ UNION deduplicates (slower)
- ✓ UNION ALL keeps duplicates (faster)
- ✓ Use UNION ALL when duplicates acceptable

6. WINDOW vs AGGREGATE FUNCTIONS



AGGREGATES

- Behavior: Collapse rows into GROUP COUNTS
- Example: Returns 1 row per department
- Rows preserved: NO - Detail rows lost
- Functions: COUNT, SUM, AVG, MAX, MIN

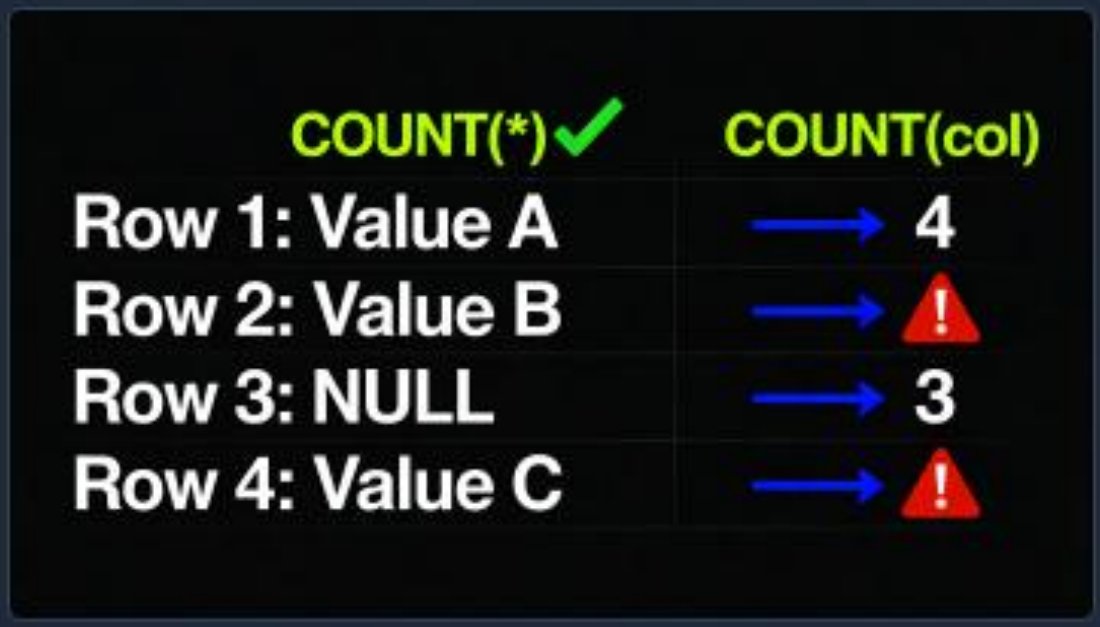
WINDOW FUNCTIONS

- Behavior: Preserve detail rows + group info
- Example: Returns all rows with dept_count repeated
- Rows preserved: YES - Detail preserved
- Functions: ROW_NUMBER, RANK, DENSE_RANK

Key Takeaways

- ✓ Aggregates: Collapse rows
- ✓ Window functions: Preserve rows
- ✓ ROW_NUMBER, RANK, DENSE_RANK are window functions

7. COUNT(*) vs COUNT(column)




```
Data: [100, 200, NULL, 150]
```

COUNT(*)

- Result: 4
- Includes NULL: YES (counts NULL)
- Counts: All rows in table
- Use: Total row count

COUNT(column)

- Result: 3
- Includes NULL: NO (ignores NULL)
- Counts: Non-NULL values only
- Use: Count of valid values

```
SUM(amount) = 450 (100+200+150, ignores NULL)
```

```
AVG(amount) = 150 (450÷3, not 450÷4)
```

Key Takeaways

- ✓ COUNT(*) counts all rows
- ✓ COUNT(col) ignores NULLs
- ✓ Other aggregates also ignore NULLs

8. INDEX SEEK vs TABLE SCAN

Index Seek

Direct jump to data

2.64 33.90 — 10/20

Table Scan

Read all pages

1036 2,370 — 26?0

INDEX SEEK

- Speed: FAST
- Behavior: Jump directly to data (like index in book)
- When: Equality/specific range on indexed column
- I/O: Minimal disk reads

TABLE/INDEX SCAN

- Speed: SLOW
- Behavior: Read all pages sequentially
- When: Large result set or non-indexed column
- I/O: Reads entire table/index

Key Takeaways

- ✓ Seek = Jump directly (FAST)
- ✓ Scan = Read everything (SLOW)
- ✓ Index on JOIN keys = performance gains

Detailed Descriptive Version

SQL DIFFERENCES VISUAL GUIDE

For Data Engineers & SQL Developers

---

1. JOIN TYPES

Figure 1: SQL Join Types - Visual Venn Diagrams (INNER, LEFT, RIGHT, FULL, CROSS)

Key Concept: Different joins return different row counts

- **INNER JOIN** = Only matching rows from both tables
- **LEFT JOIN** = All rows from left table + matches from right
- **RIGHT JOIN** = All rows from right table + matches from left
- **FULL JOIN** = All rows from both tables (MySQL: use UNION of LEFT + RIGHT)
- **CROSS JOIN** = Cartesian product (every combination)

---

2. CRITICAL: WHERE vs. ON in Outer Joins

The Deadly Mistake:

```
LEFT JOIN table2 ON t1.id = t2.id
WHERE t2.value > 100; -- ❌ Converts LEFT to INNER!
```

The Fix:

```
LEFT JOIN table2 ON t1.id = t2.id AND t2.value > 100; -- ✅ Correct
```

Why: WHERE executes AFTER join and rejects NULL rows, killing outer join behavior.

3. NULL & THREE-VALUED LOGIC

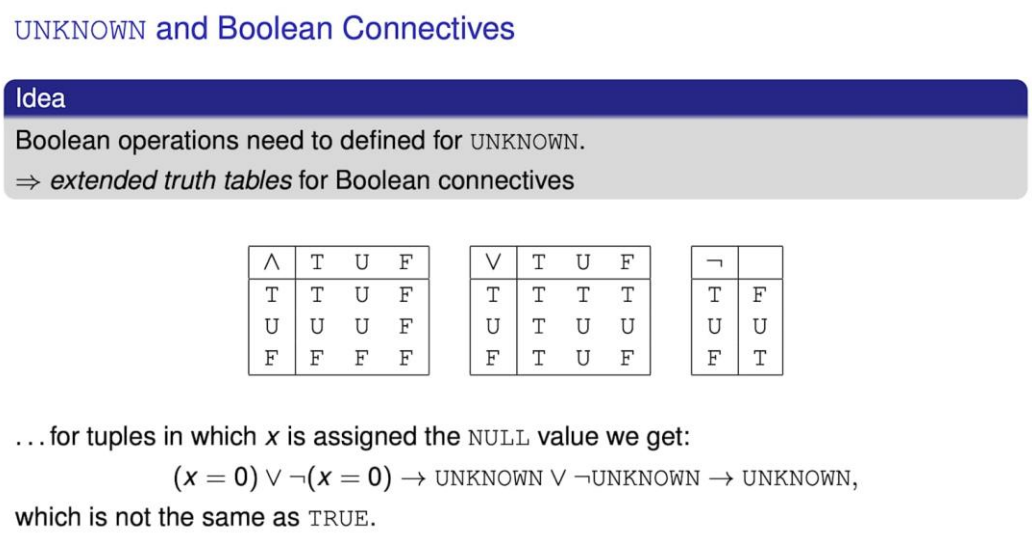


Figure 2: Three-Valued Logic (TRUE, FALSE, UNKNOWN)

Critical Rules:

Expression	Result	Why
5 = 5	TRUE	Direct match
5 = NULL	UNKNOWN	Can't compare to unknown
NULL = NULL	UNKNOWN	Both unknown
WHERE keeps:	TRUE only	FALSE and UNKNOWN both rejected

ALWAYS Use:

WHERE column IS NULL -- ✅ Correct

WHERE column = NULL -- ❌ Wrong (always FALSE)

---

4. GROUP BY vs. DISTINCT

DISTINCT	GROUP BY
Removes duplicates	Aggregates with SUM, COUNT, AVG
No aggregation	Requires aggregation functions
Faster for small sets	Use with grouping logic
SELECT DISTINCT col	SELECT col, COUNT(*) GROUP BY col

Table 1: DISTINCT vs. GROUP BY

```
-- DISTINCT: Just remove duplicates
SELECT DISTINCT country FROM customers;

-- GROUP BY: Aggregate within groups
SELECT country, COUNT(*) as count
FROM customers
GROUP BY country;
```

---

5. WHERE vs. HAVING (Query Execution Order)

Figure 3: Query Execution Pipeline: FROM → WHERE → GROUP BY → HAVING → SELECT

Execution Order:

1. FROM / JOIN (identify tables)
2. **WHERE** (filter BEFORE grouping) ← FASTER
3. GROUP BY (create groups)
4. Aggregation (SUM, COUNT, etc.)
5. **HAVING** (filter AFTER aggregation)
6. SELECT (choose columns)
7. ORDER BY (sort results)

Why It Matters:

```
-- ✅ FAST: WHERE filters first (fewer rows grouped)
SELECT dept, COUNT(
```

)FROM employeesWHERE salary > 50000GROUP BY deptHAVING COUNT() > 5;

-- ❌ SLOW: Groups all, then filters
SELECT dept, COUNT(

)FROM employeesGROUP BY deptHAVING COUNT() > 5; -- If no WHERE

---

6. WINDOW FUNCTIONS vs. AGGREGATES

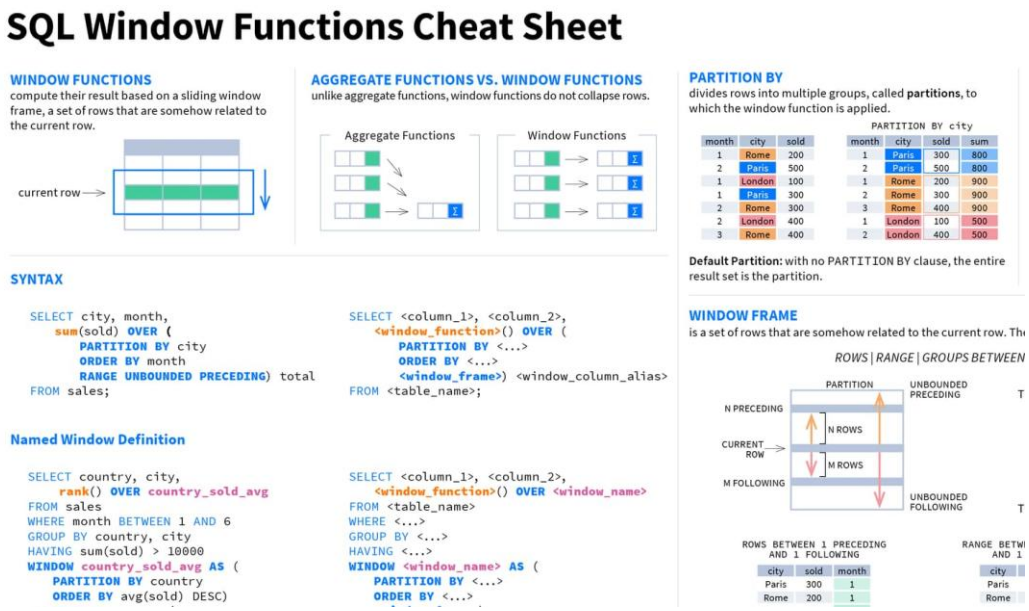


Figure 4: Window Functions with PARTITION BY, ORDER BY, and Frame Specification

Key Difference:

- **Aggregates** (GROUP BY): Collapse rows into single group rows
- **Window Functions**: Keep detail rows, add group calculations

-- Aggregate: Returns 1 row per department
SELECT dept, COUNT(*) FROM employees GROUP BY dept;

-- Window: Returns all rows with dept_count repeated
SELECT name, dept,
COUNT(*) OVER (PARTITION BY dept) as dept_count
FROM employees;

Ranking Functions:
ROW_NUMBER() -- 1, 2, 3, 4 (unique sequence)
RANK() -- 1, 2, 2, 4 (gaps on ties)
DENSE_RANK() -- 1, 2, 2, 3 (no gaps)

7. ROWS vs. RANGE Frames

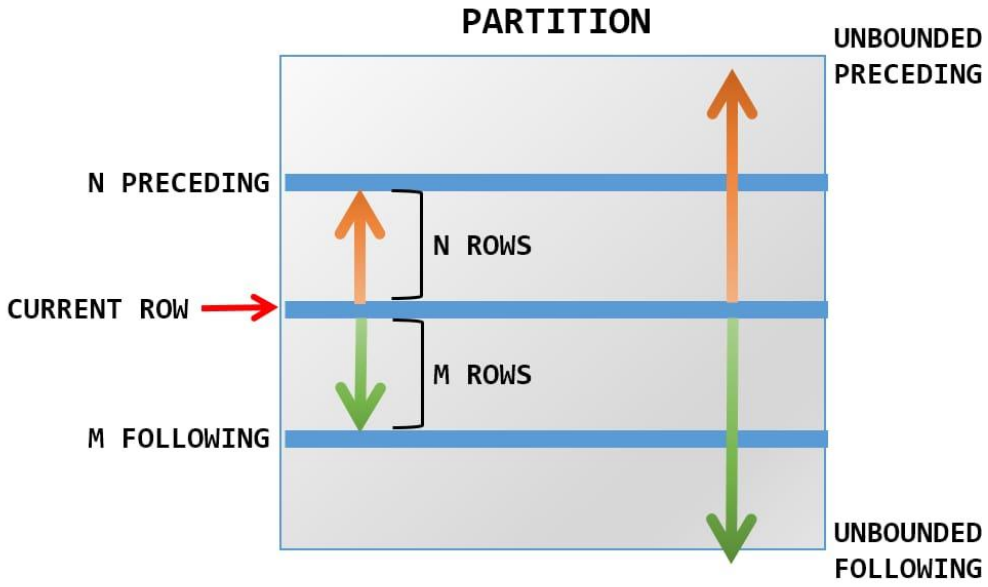


Figure 5: Window Function Frame Types: ROWS (physical) vs. RANGE (value-based)

Frame Type	Definition	Use Case
ROWS BETWEEN 1 PRECEDING AND CURRENT	Physical row count	Moving average (exact N rows)
RANGE BETWEEN INTERVAL '7 days' PRECEDING	Value-based range	Date-based windows

-- Running total (last 3 rows)
SUM(amount) OVER (ORDER BY date ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)

-- 7-day moving average
AVG(sales) OVER (ORDER BY date RANGE BETWEEN INTERVAL '7 days' PRECEDING AND CURRENT ROW)

8. SET OPERATIONS: UNION / INTERSECT / EXCEPT

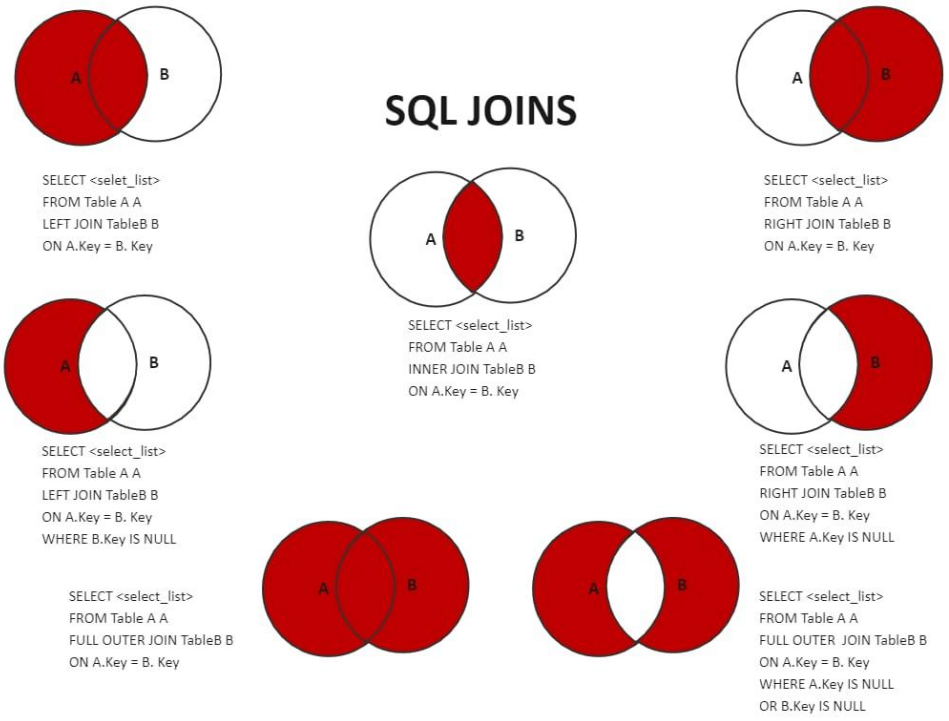


Figure 6: Set Operations Venn Diagrams (UNION, INTERSECT, EXCEPT)

Operation	Behavior	Speed
UNION	Remove duplicates	Slower (dedup)
UNION ALL	Keep duplicates	**Faster**
INTERSECT	Common rows	Medium
EXCEPT / MINUS	First not second	Medium

Performance Tip: Use UNION ALL when duplicates are acceptable.

---

9. COUNT(*) vs. COUNT(column)

Count Type	Counts	Includes NULLs?
COUNT(*)	All rows	Yes (counts NULLs)
COUNT(column)	Non-NULL values	No (ignores NULLs)
COUNT(DISTINCT column)	Unique values	No

-- Data: [100, 200, NULL, 150]
COUNT(*) -- 4 (all rows)

COUNT(amount) -- 3 (ignores NULL)
SUM(amount) -- 450 (100+200+150, ignores NULL)
AVG(amount) -- 150 (450÷3, not 450÷4)

---

10. PERFORMANCE: Index Seek vs. Scan

Figure 7: Query Optimization: Index Seek (Fast) vs. Table Scan (Slow)

Strategy	Speed	When
Index Seek	FAST	Equality/specific range on indexed column
Table/Index Scan	SLOW	Large result set or non-indexed column

-- Seek: Uses index to jump directly
SELECT * FROM employees WHERE id = 42;

-- Scan: Reads entire table/index
SELECT * FROM employees WHERE id > 42; -- May scan if low selectivity

---

11. JOIN ALGORITHMS (Optimizer Chooses)

Algorithm	Conditions	Speed
Nested Loop	Small table available	Slow for large data
Hash Join	Large unindexed tables	Fast, memory-intensive
Merge Join	Pre-sorted data	Fast if indexed

Strategy: Ensure indexes on join keys; optimizer will pick best algorithm.

---

12. DATA TYPES MATTER

String Types

Type	Storage	Use
CHAR(n)	Fixed, padded	Codes, fixed IDs
VARCHAR(n)	Variable	Names, emails
TEXT	Large variable	Documents, content

Numeric Types

INT / BIGINT -- Exact integers
NUMERIC(10,2) -- Exact decimals (use for money!)
FLOAT / DOUBLE -- Approximate (AVOID for money)

Date/Time

DATE -- YYYY-MM-DD only
TIMESTAMP -- Date + time
TIMESTAMP WITH TZ -- Date + time + timezone

---

13. MUST-KNOW DIFFERENCES (Interview Prep)

1. **INNER vs. LEFT JOIN** — Rows from both vs. all from left + matches
2. **WHERE vs. ON** — After join vs. during join (critical for outer joins!)
3. **NULL = NULL → UNKNOWN** — Three-valued logic, use IS NULL

4. **GROUP BY vs. DISTINCT** — Aggregation vs. deduplication
5. **WHERE vs. HAVING** — Pre-group vs. post-group filters
6. **UNION vs. UNION ALL** — Deduplicate vs. keep (speed difference!)
7. **Aggregate vs. Window** — Collapse vs. preserve detail rows
8. **ROWS vs. RANGE** — Physical count vs. value-based frame
9. **COUNT(*) vs. COUNT(col)** — All rows vs. non-NULL
10. **Index Seek vs. Scan** — Direct vs. full read (performance critical!)

---

14. QUICK REFERENCE CHEAT SHEET

Figure 8: SQL Cheat Sheet - Common Patterns & Syntax

---

15. KEY TAKEAWAYS

✓ **Master JOINs** — INNER, LEFT, RIGHT, FULL; understand NULL behavior
✓ **WHERE vs. HAVING** — Placement determines pre- vs. post-aggregation filtering
✓ **NULL Handling** — Three-valued logic is non-obvious; always use IS NULL
✓ **Window Functions** — Essential for analytics; preserves detail rows
✓ **Performance** — Filter early (WHERE), use indexes, understand seek vs. scan
✓ **Set Operations** — UNION ALL faster than UNION when duplicates acceptable
✓ **Data Types** — DECIMAL for money, TIMESTAMP for dates

Resources & Further Learning

[1] ISO/IEC 9075-1:2023. Database Language SQL – Part 1: Framework and Overview. ISO Standard.

[2] Garcia-Molina, H., Ullman, J. D., & Widom, J. (2008). *Database Systems: The Complete Book* (2nd ed.). Prentice Hall.

[3] Modern SQL Contributors. (2024). Three-Valued Logic (3VL). https://modern-sql.com/concept/three-valued-logic

[4] PostgreSQL Documentation. (2025). Window Functions. Official Documentation.

[5] Microsoft SQL Server Documentation. (2025). Joins and Query Optimization.
https://learn.microsoft.com/sql