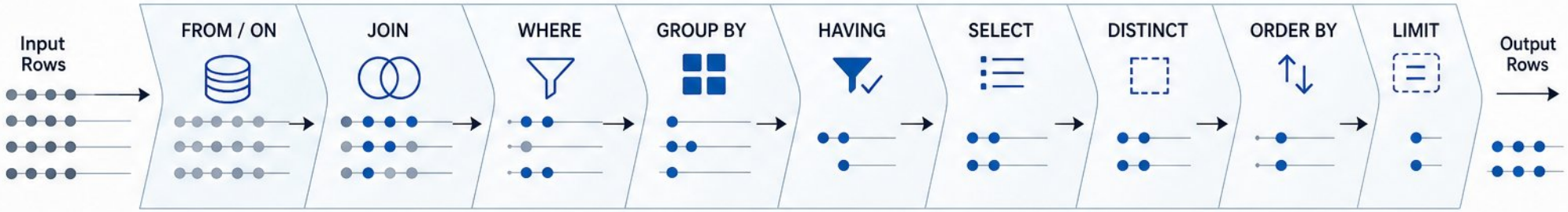
# SQL Database Query Examples:

## Patterns, Strengths, and Pitfalls

- **Goal:** build durable instincts for picking the right SQL pattern, not memorizing syntax.
- **Roadmap:** how queries run, JOIN patterns, aggregation, subqueries vs CTEs vs windows.
- **Then** pagination, NULL and casting pitfalls, dialect portability, review checklist, hands-on lab.
- **Assumed baseline:** relational model plus basic SELECT, WHERE, and JOIN.
- **Every** example is read as schema, query, reasoning, result shape, risk.
- **Each** pattern ships with its strength and the pitfall it guards against.

# How a Query Actually Runs: Reading an Example End to End

```sql
SELECT c.name AS customer_name, COUNT(o.id) AS order_count
FROM customers c
JOIN orders o ON o.customer_id = c.id
WHERE c.active = 1
GROUP BY c.name
HAVING COUNT(o.id) > 0
ORDER BY order_count DESC
LIMIT 10
```



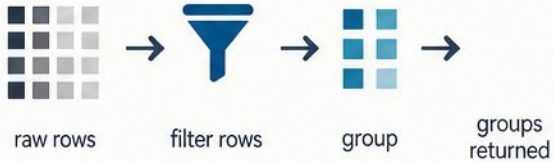

- Logical order: FROM and ON → JOIN → WHERE → GROUP BY → HAVING → SELECT → DISTINCT → ORDER BY → LIMIT
- Written order differs; that gap explains most confusing beginner results
- SELECT aliases work in ORDER BY, rarely in WHERE: SELECT runs after WHERE
- Execution plan basics: scans, join methods, sorts, aggregation nodes
- Optimizers reorder work physically; logical order stays the correctness model

# WHERE vs HAVING, Aliases, and Other Ordering Traps

- WHERE filters individual rows before grouping; HAVING filters groups after aggregation
- **Rule:** raw-row reducible conditions go in WHERE; COUNT, SUM, AVG, MIN, MAX go in HAVING
- ORDER BY usually accepts SELECT aliases; WHERE generally does not
- MySQL allows non-aggregated columns and HAVING references the standard forbids
- Column positions in ORDER BY are deprecated; PostgreSQL allows output names in GROUP BY
- Pitfall: visual-order queries return plausible but semantically wrong results

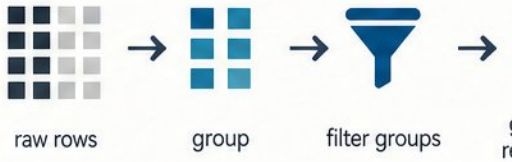
## Filter Raw Rows with WHERE (Good Pattern)

```sql
SELECT
  department,
  COUNT(*) AS employee_count
FROM employees
WHERE active = 1        -- filter rows first
GROUP BY department
ORDER BY employee_count DESC;
```




## Filter Groups with HAVING (Good Pattern)

```sql
SELECT
  department,
  COUNT(*) AS employee_count
FROM employees
GROUP BY department
HAVING COUNT(*) > 1       -- filter groups
ORDER BY employee_count DESC;
```




## Pitfall: Visual-Order Variant (Wrong)

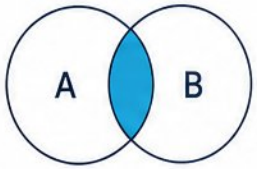
```sql
SELECT
  department,
  COUNT(*) AS employee_count
FROM employees
WHERE COUNT(*) > 1        -- WRONG: aggregate in WHERE
GROUP BY department
ORDER BY employee_count DESC;
```

**Visually similar, semantically wrong.**

This filters before grouping, not after aggregation.

# JOIN Patterns: INNER, LEFT, RIGHT, FULL, CROSS, and Self-Joins

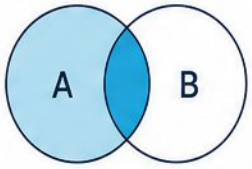
### INNER JOIN




Only the overlap (matches)

```
SELECT ...
FROM A
INNER JOIN B
ON A.key = B.key;
```

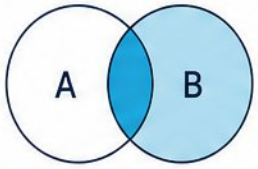
### LEFT JOIN




All rows from A, matched rows from B

```
SELECT ...
FROM A
LEFT JOIN B
ON A.key = B.key;
```

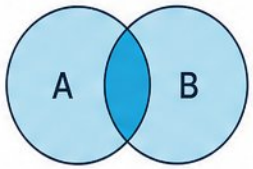
### RIGHT JOIN




All rows from B, matched rows from A

```
SELECT ...
FROM A
RIGHT JOIN B
ON A.key = B.key;
```

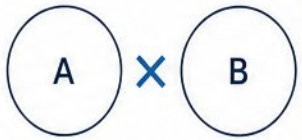
### FULL JOIN




All rows from A and B

```
SELECT ...
FROM A
FULL JOIN B
ON A.key = B.key;
```

### CROSS JOIN




Cartesian product (all combinations)

```
SELECT ...
FROM A
CROSS JOIN B;
```

### SELF-JOIN

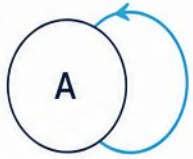



Table joined to itself (e.g., hierarchies)

```
SELECT ...
FROM A AS parent
JOIN A AS child
ON parent.id = child.pid;
```

- Pick the join type for its set semantics, not style
- INNER: intersection; LEFT: preserve left rows; FULL: preserve both sides
- CROSS: deliberate Cartesian product; self-joins model hierarchies
- MySQL treats JOIN, CROSS JOIN, and INNER JOIN as equivalent
- Prefer LEFT JOIN for readability when chaining outer joins
- Pitfall: row multiplication when the join key is not unique on the many-side

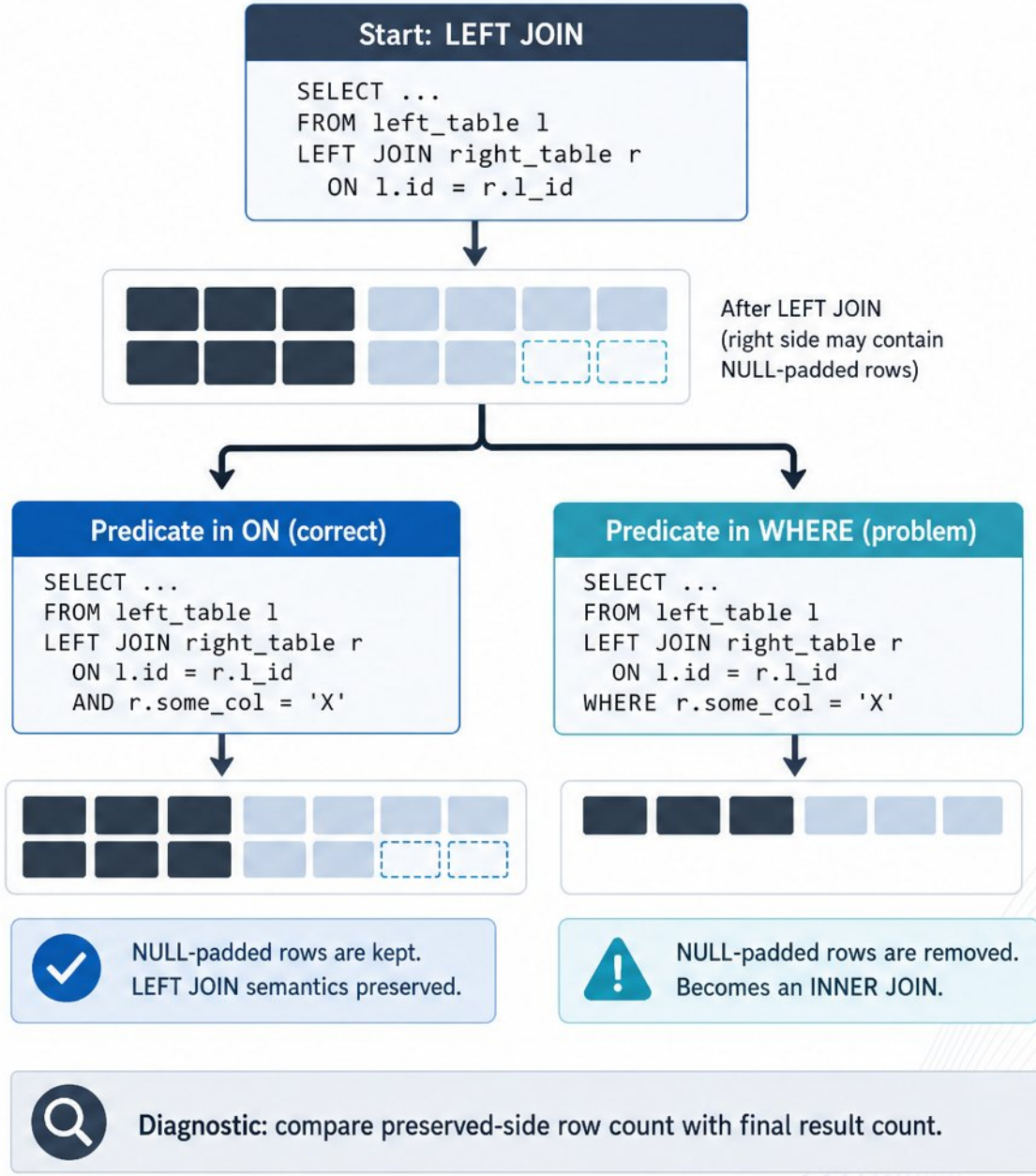
## MySQL equivalence

```
SELECT ...
FROM A JOIN B ON A.key = B.key;
-----------------------------------
SELECT ...
FROM A CROSS JOIN B ON A.key = B.key;
-----------------------------------
SELECT ...
FROM A INNER JOIN B ON A.key = B.key;
```

These forms return the same result set in MySQL.

# Left Join Filters: The Accidental Inner Join

- **ON** filters during the join; **WHERE** filters after the join completes
- **INNER JOIN: ON** and **WHERE** predicates return the same rows
- **LEFT JOIN:** a right-side **WHERE** predicate removes NULL-padded rows
- That silently converts the LEFT JOIN into an **INNER JOIN**
- Fix: move the right-side predicate into the **ON** clause
- Valid exception: **WHERE** right_table.key IS NULL finds unmatched rows (anti-join)
- Diagnostic: compare preserved-side row count with final result count



# Aggregation and GROUP BY: Trustworthy Summaries

- Define the grain first: what does one output row represent?
- COUNT(*) counts rows; COUNT(column) silently skips NULL values
- Conditional aggregation with CASE builds cross-tab summaries, no pivot needed
- MySQL's ONLY_FULL_GROUP_BY blocks non-grouped columns; know dialect rules
- Pitfall: wrong grain yields plausible but false totals
- Verify: recompute summary at coarser grain and compare totals

## Grain defines the meaning of each row

**Grouping keys (the grain)**

- Dimension A
- Dimension B

**Summary Table (one row per grain)**

| Dimension A | Dimension B | COUNT(*) | COUNT(column) |
|---|---|---|---|
| | | | |
| | | | |
| | | | |

## Cross-tab via conditional aggregation (no pivot needed)

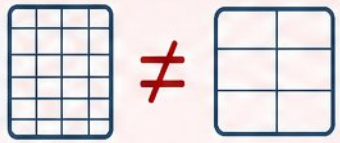
```
SELECT
  Dimension A,
  Dimension B,
  COUNT(*) AS row_count,
  SUM(CASE WHEN Category = 'X' THEN 1 ELSE 0 END) AS cat_x_count,
  SUM(CASE WHEN Category = 'Y' THEN 1 ELSE 0 END) AS cat_y_count
FROM your_table
GROUP BY Dimension A, Dimension B;
```

**Category (conditional counts)**

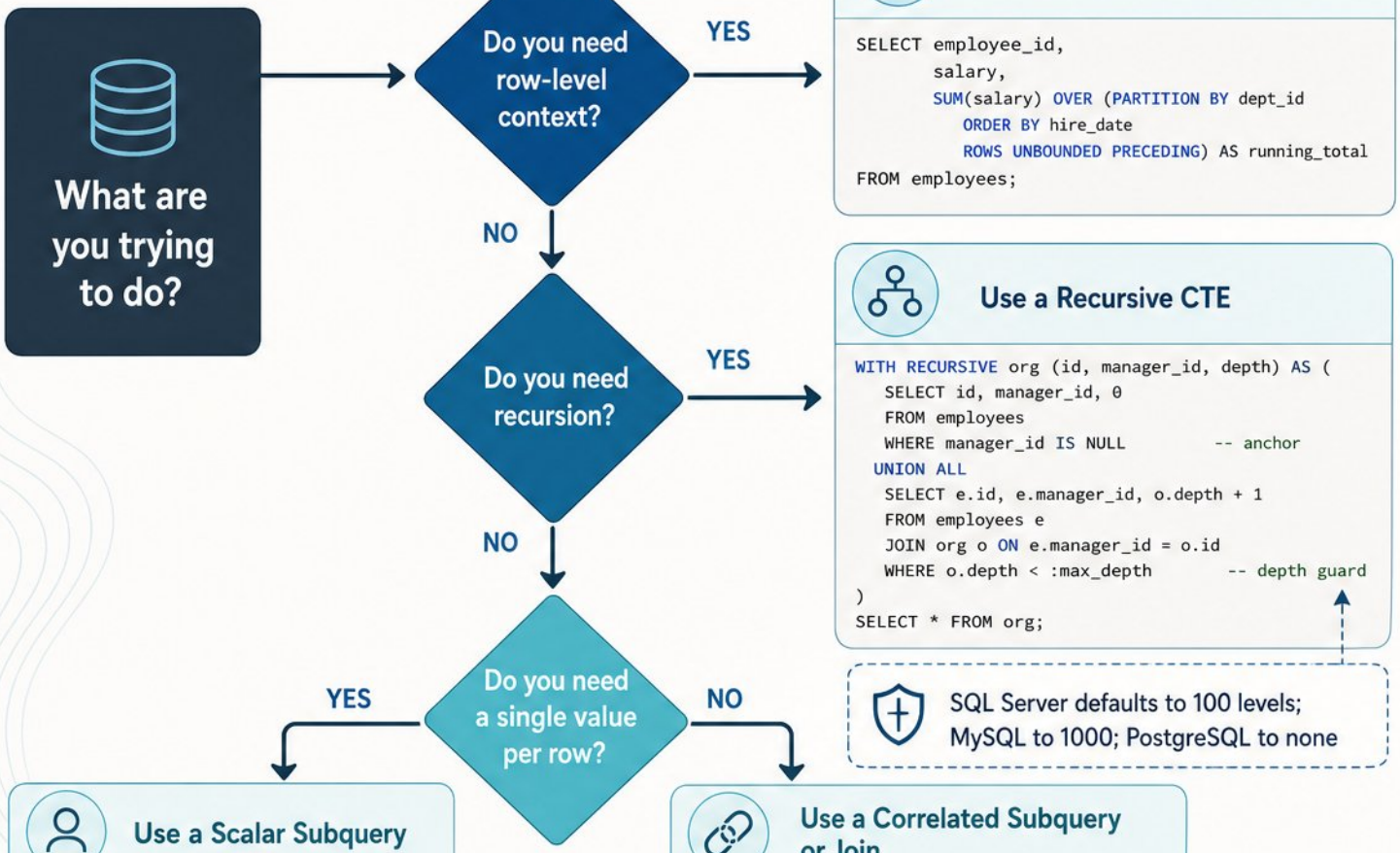
| Dimension A | Dimension B | cat_x_count | cat_y_count |
|---|---|---|---|
| | | | |
| | | | |
| | | | |

**Mismatched totals indicate wrong grain.**

Recompute at a coarser grain and compare totals to verify.



# Subqueries vs CTEs vs Window Functions: Choosing the Right Tool




### Use a Window Function

```
SELECT employee_id,
       salary,
       SUM(salary) OVER (PARTITION BY dept_id
          ORDER BY hire_date
          ROWS UNBOUNDED PRECEDING) AS running_total
FROM employees;
```

### Use a Recursive CTE

```
WITH RECURSIVE org (id, manager_id, depth) AS (
   SELECT id, manager_id, 0
   FROM employees
   WHERE manager_id IS NULL          -- anchor
  UNION ALL
   SELECT e.id, e.manager_id, o.depth + 1
   FROM employees e
   JOIN org o ON e.manager_id = o.id
   WHERE o.depth < :max_depth        -- depth guard
)
SELECT * FROM org;
```

SQL Server defaults to 100 levels; MySQL to 1000; PostgreSQL to none

### Use a Scalar Subquery

```
SELECT e.name,
     (SELECT MAX(salary)
      FROM employees e2
      WHERE e2.dept_id = e.dept_id) AS max_salary
FROM employees e;
```

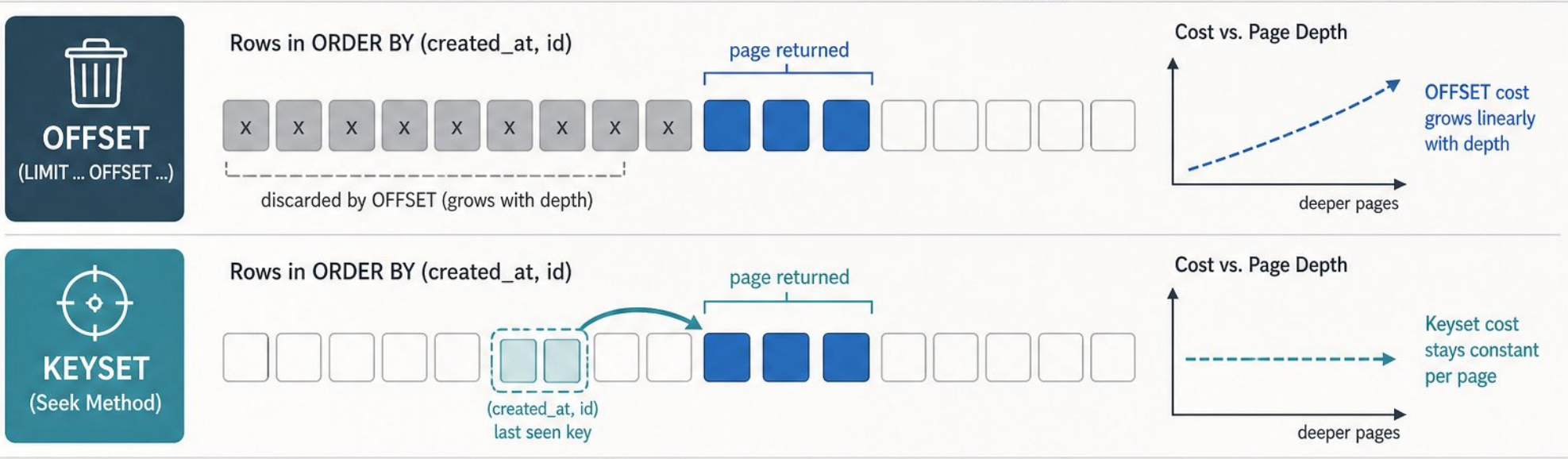
### Use a Correlated Subquery or Join

```
SELECT e.name, d.avg_salary
FROM employees e
JOIN (SELECT dept_id, AVG(salary) AS avg_salary
      FROM employees
      GROUP BY dept_id) d
  ON e.dept_id = d.dept_id;
```

- Scalar subqueries read well but may execute once per row
- Correlated subqueries often rewrite cleanly as joins or window functions
- CTEs add structure and enable recursion; materialization varies by engine
- Window functions keep row context: running totals, ranking, LAG/LEAD
- Recursive CTEs need UNION ALL, an anchor, and a depth guard
- SQL Server defaults to 100 levels; MySQL to 1000; PostgreSQL to none

# Pagination, Ordering, and Determinism

- LIMIT and OFFSET without deterministic ORDER BY: pages drift
- OFFSET scans and discards rows; cost grows linearly with depth
- Keyset pagination seeks past the last key: constant cost per page
- Total order needs a unique tiebreaker: (created_at, id)
- Missing composite index on the ORDER BY tuple kills the keyset win
- Dialects: LIMIT n OFFSET m; OFFSET m ROWS FETCH NEXT n ROWS ONLY




## Dialect Examples

**PostgreSQL**

```
SELECT *
FROM t
ORDER BY created_at, id
LIMIT n OFFSET m;
```

**MySQL**

```
SELECT *
FROM t
ORDER BY created_at, id
LIMIT n OFFSET m;
```

**SQL Server**

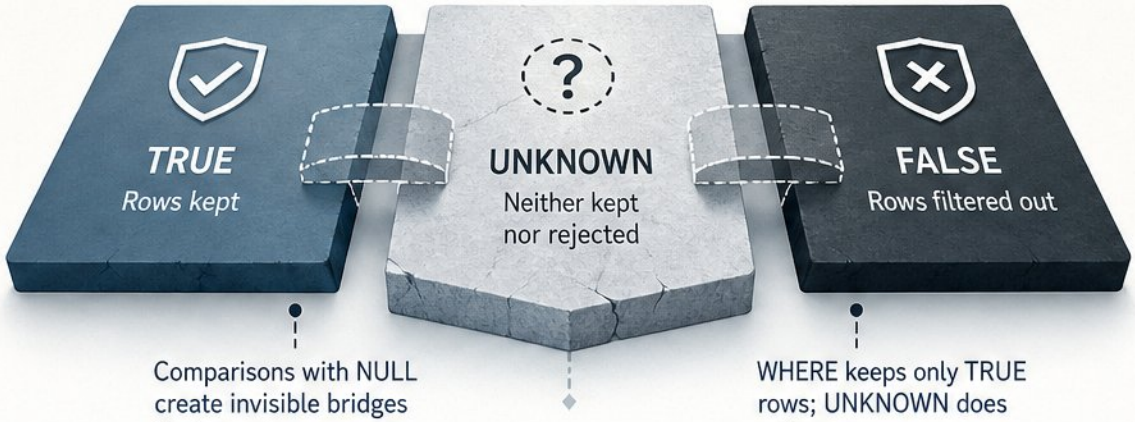
```
SELECT *
FROM t
ORDER BY created_at, id
OFFSET m ROWS
FETCH NEXT n ROWS ONLY;
```

**Oracle**

```
SELECT *
FROM t
ORDER BY created_at, id
OFFSET m ROWS
FETCH NEXT n ROWS ONLY;
```

# NULLs, Implicit Casts, and Duplicate Rows

## THREE-VALUED LOGIC LANDSCAPE




### GOOD PATTERN

```sql
-- NULL-aware comparison
SELECT *
FROM orders o
WHERE o.ship_date IS NULL;

-- Explicit, sargable cast
SELECT *
FROM orders o
WHERE o.order_id = CAST('123' AS INT);

-- Preserve grain; fix joins
SELECT o.id, o.customer_id
FROM orders o
JOIN customers c
  ON o.customer_id = c.id;
```

vs.

### MIRRORED PITFALL

```sql
-- UNKNOWN filtered out
SELECT *
FROM orders o
WHERE o.ship_date = NULL; -- always UNKNOWN

-- Implicit cast on column
SELECT *
FROM orders o
WHERE CAST(o.order_id AS CHAR) = '123';
-- index on order_id ignored

-- Masking duplicates with DISTINCT
SELECT DISTINCT o.id, o.customer_id
FROM orders o
JOIN customers c
  ON o.customer_id = c.id; -- hides issue
```

- Three-valued logic: comparisons with NULL yield UNKNOWN; WHERE keeps only TRUE rows
- Use `IS NULL`, `IS NOT NULL`, or `IS [NOT] DISTINCT FROM` for NULL-aware comparison
- `IS NOT DISTINCT FROM` returns TRUE for NULL vs NULL; MySQL uses `<=>`
- Implicit casts silently drop index usage; make casts explicit and sargable
- **DISTINCT** as a band-aid hides earlier join or grain errors — fix the source
- Compare row counts after each stage; first divergence usually holds the bug

# Portable vs Dialect-Specific Query Patterns

- SQL-92 baseline travels well: joins, CASE, CAST, COALESCE, basic string and date functions.
- Divergence hotspots: pagination, string aggregation, date arithmetic, booleans, identifier quoting.
- Date functions do not translate one-to-one; keep functions off indexed columns for sargability.
- CONCAT NULLs differ: MySQL returns NULL, PostgreSQL ignores, SQL Server returns empty string.
- Row limits: LIMIT (PostgreSQL/MySQL), TOP or OFFSET/FETCH (SQL Server), FETCH FIRST plus legacy ROWNUM (Oracle).

| | PostgreSQL | MySQL | SQL Server | Oracle |
|---|---|---|---|---|
| **Pagination** | `SELECT * FROM t ORDER BY id LIMIT 10 OFFSET 20;` | `SELECT * FROM t ORDER BY id LIMIT 20, 10;` | `SELECT * FROM t ORDER BY id OFFSET 20 ROWS FETCH NEXT 10 ROWS ONLY;` | `SELECT * FROM t ORDER BY id FETCH FIRST 10 ROWS ONLY; -- or legacy ROWNUM <= 10` |
| **String Aggregation** | `SELECT string_agg (name, ',') FROM t;` | `SELECT GROUP_CONCAT (name SEPARATOR ',') FROM t;` | `SELECT STRING_AGG (name, ',') FROM t;` | `SELECT LISTAGG (name, ',') WITHIN GROUP (ORDER BY name) FROM t;` |
| **Date Arithmetic** | `SELECT order_date + INTERVAL '7 days' FROM orders;` | `SELECT DATE_ADD (order_date, INTERVAL 7 DAY) FROM orders;` | `SELECT DATEADD (DAY, 7, order_date) FROM orders;` | `SELECT order_date + INTERVAL '7' DAY FROM orders;` |
| **Boolean Types** | `SELECT is_active FROM users WHERE is_active = TRUE;` | `SELECT is_active FROM users WHERE is_active = 1;` | `SELECT is_active FROM users WHERE is_active = 1; -- BIT` | `No native boolean; use NUMBER(1) or CHAR(1) and check = 1;` |
| **Identifier Quoting** | `SELECT "UserName" FROM "Order";` | `SELECT `UserName` FROM `Order`;` | `SELECT [UserName] FROM [Order];` | `SELECT "UserName" FROM "Order";` |
| **Row Limits** | `LIMIT n OFFSET m` | `LIMIT m, n` | `TOP (n) or OFFSET m ROWS FETCH NEXT n ROWS ONLY` | `FETCH FIRST n ROWS ONLY or ROWNUM <= n` |

# Worked Cross-Dialect Rewrite

PostgreSQL · MySQL · SQL Server · Oracle

- Task: ten highest-revenue UTC days, paid orders, trailing 30 days
- Ties broken by earlier date; NULL amounts treated as zero via COALESCE
- PostgreSQL: `date_trunc('day', ordered_at)`, `INTERVAL '30 days'`, `LIMIT 10`
- MySQL: `CAST(ordered_at AS DATE)`, `DATE_SUB(CURRENT_TIMESTAMP, INTERVAL 30 DAY)`
- SQL Server: `DATEADD(day, -30, ...)` with `SELECT TOP (10)`, no `LIMIT`

## PostgreSQL

**CONCEPTUAL OPERATION**

1. Bucket to day (UTC)
2. Filter: paid orders, trailing 30 days
3. Sum revenue with NULLs as zero
4. Order by revenue desc, earlier date
5. Return top 10

```
SELECT date_trunc('day', ordered_at) AS day_utc,
       COALESCE(SUM(amount), 0) AS revenue
FROM orders
WHERE status = 'paid'
  AND ordered_at >= (CURRENT_TIMESTAMP AT TIME ZONE 'UTC') - INTERVAL '30 days'
GROUP BY 1
ORDER BY revenue DESC, day_utc ASC
LIMIT 10;
```

## MySQL

**CONCEPTUAL OPERATION**

1. Bucket to day (UTC)
2. Filter: paid orders, trailing 30 days
3. Sum revenue with NULLs as zero
4. Order by revenue desc, earlier date
5. Return top 10

```
SELECT CAST(ordered_at AS DATE) AS day_utc,
       COALESCE(SUM(amount), 0) AS revenue
FROM orders
WHERE status = 'paid'
  AND ordered_at >= DATE_SUB(CURRENT_TIMESTAMP, INTERVAL 30 DAY)
GROUP BY 1
ORDER BY revenue DESC, day_utc ASC
LIMIT 10;
```

## SQL Server

**CONCEPTUAL OPERATION**

1. Bucket to day (UTC)
2. Filter: paid orders, trailing 30 days
3. Sum revenue with NULLs as zero
4. Order by revenue desc, earlier date
5. Return top 10

```
SELECT TOP (10) CAST(ordered_at AS date) AS day_utc,
       COALESCE(SUM(amount), 0) AS revenue
FROM orders
WHERE status = 'paid'
  AND ordered_at >= DATEADD(day, -30, CURRENT_TIMESTAMP)
GROUP BY CAST(ordered_at AS date)
ORDER BY revenue DESC, day_utc ASC;
```

**Name the operation first, then map it to the engine**

# Query Review Checklist: Strengths, Risks, and Testing

1. Pre-flight: confirm grain, join keys, filter placement, NULL handling, deterministic ordering, row limits.
2. Compare estimated versus actual rows; large gaps reveal weak predicates or stale statistics.
3. Test edge cases: empty set, single row, repeated keys, NULLs on both sides, extreme dates.
4. **Read EXPLAIN** fast: scan type, join method, sort and aggregation nodes, row estimates.
5. **Safety nets:** wrap exploration in a transaction or read-only replica, cap rows.
6. **Document intent:** business question, grain, and assumptions about keys or data quality.

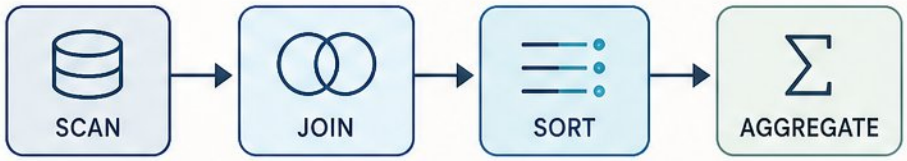
## EDGE CASE TEST MATRIX


## EXPLAIN (ABBREVIATED)

```
EXPLAIN SELECT o.order_id, c.customer_name, o.order_date
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
WHERE o.order_date >= DATE '2024-01-01'
ORDER BY o.order_date;

 Seq Scan on orders o     (rows=...)
 └─ Hash Join              (rows=...)
     └─ Index Scan on customers c  (rows=...)
 Sort                      (rows=...)
 Aggregate                 (rows=...)
```

## EXECUTION PLAN AT A GLANCE



# Practice Lab: Apply the Patterns

| EXERCISE | BROKEN / PITFALL SHAPE | YOUR REWRITE (FILL IN) | GOOD PATTERN SHAPE |
|---|---|---|---|
| **1** **Exercise 1:** move a right-side predicate from WHERE into ON to save the LEFT JOIN | `SELECT o.id, c.name`<br>`FROM orders o`<br>`LEFT JOIN customers c`<br>`  ON o.customer_id = c.id`<br>`WHERE c.status = 'active';` | | `SELECT o.id, c.name`<br>`FROM orders o`<br>`LEFT JOIN customers c`<br>`  ON o.customer_id = c.id`<br>`  AND c.status = 'active';` |
| **2** **Exercise 2:** replace OFFSET paging with keyset seek on a (sort_column, id) tuple | `SELECT id, sort_column, other_cols`<br>`FROM items`<br>`ORDER BY sort_column, id`<br>`OFFSET :offset ROWS`<br>`FETCH NEXT :limit ROWS ONLY;` | | `SELECT id, sort_column, other_cols`<br>`FROM items`<br>`WHERE (sort_column, id) > (:last_sort,`<br>`  :last_id)`<br>`ORDER BY sort_column, id`<br>`FETCH NEXT :limit ROWS ONLY;` |
| **3** **Exercise 3:** refactor a correlated subquery into a window function; compare plans | `SELECT o.id, o.customer_id,`<br>`  (SELECT MAX(p.created_at)`<br>`  FROM payments p`<br>`  WHERE p.order_id = o.id) AS last_pay_at`<br>`FROM orders o;` | | `SELECT o.id, o.customer_id,`<br>`  MAX(p.created_at) OVER (PARTITION BY o.id)`<br>`  AS last_pay_at`<br>`FROM orders o`<br>`LEFT JOIN payments p ON p.order_id = o.id;` |
| **4** **Exercise 4:** fix a GROUP BY grain error that produced plausible but false totals | `SELECT customer_id, SUM(amount) AS total`<br>`FROM orders`<br>`GROUP BY customer_id;` | | `SELECT customer_id, order_date,`<br>`  SUM(amount) AS total`<br>`FROM orders`<br>`GROUP BY customer_id, order_date;` |
| **5** **Exercise 5:** test a NULL-safe IS NOT DISTINCT FROM comparison with NULLs on both sides | `-- comparisons with NULL fail`<br>`SELECT *`<br>`FROM t a`<br>`JOIN t b ON a.key = b.key;` | | `SELECT *`<br>`FROM t a`<br>`JOIN t b`<br>`  ON a.key IS NOT DISTINCT FROM b.key;` |

**Debrief:**
name the pattern, its strength, and the pitfall it guards against

**Pattern** .....
.....

**Strength** .....
.....

**Pitfall it guards against** .....
.....

# Wrap-Up: Instincts Over Memorized Syntax

1. Five habits: state the grain first, choose joins by semantics, place filters deliberately
2. Make ordering deterministic, and always test NULL edge cases
3. Prefer readable, portable patterns; document dialect-specific trade-offs
4. Every pattern has a mirror pitfall — learn them as pairs
5. Next: run the checklist on one real query and rerun the lab

| Good Pattern | VS | Mirrored Pitfall |
|---|---|---|
| `SELECT customer_id, SUM(amount) FROM orders GROUP BY customer_id;` | ⟷ | `SELECT * FROM orders;` |
| `SELECT c.id, o.id FROM customers c LEFT JOIN orders o ON o.customer_id = c.id;` | ⟷ | `SELECT c.id, o.id FROM customers c JOIN orders o ON o.customer_id = c.id;` |
| `SELECT * FROM orders WHERE status = 'shipped';` | ⟷ | `SELECT * FROM orders WHERE status = 'shipped' OR status IS NULL;` |
| `SELECT * FROM events ORDER BY event_time, id;` | ⟷ | `SELECT * FROM events ORDER BY event_time;` |
| `SELECT * FROM events WHERE user_id IS NULL;` | ⟷ | `SELECT * FROM events WHERE user_id = NULL;` |

PersonWise.

PersonWise • AI digital-human course platform

Scan the code or click anywhere to watch this course live, with voice Q&A

personwise.ai/t/d3aa84f2/public