CSL303/MAL505 — Database Management Systems

Module I: Relational Foundations & SQL
Lecture 9a: Derived Tables, CTEs, NULLs & Views

Amit Kumar Dhar

IIT Bhilai

Agenda

- Derived tables & CTEs (`WITH`)
- NULLs & three-valued logic
- CASE / COALESCE / NULLIF
- Views

Performance intuition for subqueries

- **Uncorrelated** subquery: usually evaluated **once** — cheap.
- **Correlated** subquery: logically **per outer row** — can be slow on big tables unless the optimizer rewrites it to a join/semijoin.
- `EXISTS` can **short-circuit** (stop at the first match) — often faster than `COUNT(*) > 0`.
- We quantify this with `EXPLAIN` in Week 10; write clearly first, tune later.

Lateral joins (a glimpse)

- A LATERAL subquery in FROM can reference earlier FROM items — like a correlated subquery that returns a **table**.

```
-- PostgreSQL: the 2 most recent orders per customer
SELECT c.name, o.order_id, o.order_date
FROM   customers c
CROSS JOIN LATERAL (
    SELECT order_id, order_date FROM orders
    WHERE cust_id = c.cust_id ORDER BY order_date DESC LIMIT 2) o;
```

- Powerful for ”top-N per group” without window functions (Postgres).

Choosing your tool

Need	Reach for
”does a match exist?”	`EXISTS` / `NOT EXISTS`
compare to an aggregate	scalar subquery
name/reuse a step	CTE (`WITH`)
hierarchy / graph	recursive CTE
”for all”	double `NOT EXISTS`
per-group top-N	window function / `LATERAL`

Common mistakes

- `NOT IN` with NULLs → empty result. Prefer `NOT EXISTS`.
- Scalar subquery returning >1 row → error.
- Forgetting to **alias** a derived table.
- Deeply nested subqueries that should be **CTEs**.
- Recursive CTE without a terminating condition (`UNION` de-dup or a depth cap).

IN vs EXISTS — which to prefer

- **`EXISTS`** short-circuits (stops at first match) and is **NULL-safe**.
- **`IN`** with a literal list is fine; **`IN (subquery)`** is fine when the subquery can't produce NULLs.
- **`NOT IN (subquery)`** is the dangerous one — prefer **`NOT EXISTS`**.
- Modern optimizers often compile all three to the same semijoin/antijoin plan — so favour the **clearest, safest** form.

CTE materialization note

- Historically, PostgreSQL treated CTEs as an **optimization fence** (materialized once). Since PG12, simple CTEs may be **inlined**.
- Force behaviour explicitly when needed:

```
WITH x AS MATERIALIZED (...) ...      -- compute once, reuse
WITH x AS NOT MATERIALIZED (...) ...  -- allow inlining into the outer query
```

- SQLite similarly may inline; don’t assume a CTE is always a barrier.

Recursive CTE: guarding against cycles

- Cyclic data (A requires B requires A) can loop forever.
- Defenses:
  - use `UNION` (de-duplicates) rather than `UNION ALL`;
  - carry a **depth** counter and stop past a limit;
  - carry a **visited path** and skip nodes already seen.

```
WITH RECURSIVE r(id, depth) AS (
  SELECT course_id, 1 FROM prereq
  UNION ALL
  SELECT p.prereq_id, r.depth+1 FROM r JOIN prereq p ON p.course_id = r.id
  WHERE r.depth < 10)
SELECT DISTINCT id FROM r;
```

NULL: three meanings, one value

- **Unknown** — we don't know the customer's city.
- **Not applicable** — an ungraded enrollment has no grade.
- **Missing** — data wasn't recorded.
- SQL uses a single `NULL` for all three, which is why logic gets subtle.

Three-valued logic (3VL)

- Predicates can be **TRUE**, **FALSE**, or **UNKNOWN**.
- Any comparison with NULL yields **UNKNOWN**:
  - `NULL = NULL` → UNKNOWN
  - `NULL <> 5` → UNKNOWN
  - `5 > NULL` → UNKNOWN
- Only rows where **WHERE** is **TRUE** are returned — UNKNOWN rows are dropped.

AND / OR / NOT with UNKNOWN

A	B	A AND B	A OR B
T	U	U	T
F	U	F	U
U	U	U	U

- `NOT UNKNOWN` = UNKNOWN.
- Note `TRUE OR UNKNOWN = TRUE`, `FALSE AND UNKNOWN = FALSE` — the known value can still decide the result.

Testing for NULL

```sql
WHERE grade IS NULL          -- correct
WHERE grade IS NOT NULL      -- correct
WHERE grade = NULL           -- WRONG: always UNKNOWN -> no rows
```

- **Never** use `= NULL` or `<> NULL`.
- `IS [NOT] DISTINCT FROM` (Postgres) compares treating NULL as a value.

NULLs surprise: the count mismatch

```
SELECT COUNT(*)      FROM takes;   -- all rows
SELECT COUNT(grade)  FROM takes;   -- fewer: NULL grades skipped
```

- Also: `SELECT ... WHERE grade = 'A' OR grade <> 'A'` misses **NULL-grade rows** (both sides UNKNOWN). Intuition says ”all rows”; SQL says ”only non-NULL.”

COALESCE — first non-NULL

```sql
SELECT name, COALESCE(city, 'Unknown') AS city
FROM   customers;
```

- `COALESCE(a, b, c)` returns the first argument that isn't NULL.
- Great for **default display values** and for turning `SUM(...)` NULLs into 0: `COALESCE(SUM(x), 0)`.

NULLIF and its uses

```sql
SELECT NULLIF(stock, 0) FROM products;    -- turn 0 into NULL
SELECT revenue / NULLIF(orders, 0) AS avg_order   -- avoid divide-by-zero
FROM   stats;
```

- `NULLIF(a, b)` returns NULL if `a = b`, else `a`.
- Classic trick: guard division by zero ($\div$ NULL $\rightarrow$ NULL, not an error).

CASE — conditional expressions

```
SELECT prod_name, price,
       CASE WHEN price < 1000 THEN 'budget'
            WHEN price < 3000 THEN 'mid'
            ELSE 'premium' END AS tier
FROM   products;
```

- Evaluated top-to-bottom; first true branch wins.
- Without ELSE, unmatched rows get NULL.

Searched vs simple CASE

```
-- simple CASE: compare one expression to values
CASE status WHEN 'completed' THEN 1 WHEN 'shipped' THEN 2 ELSE 3 END

-- searched CASE: arbitrary conditions
CASE WHEN price > 2000 AND stock = 0 THEN 'restock' ELSE 'ok' END
```

- Use `CASE` in `SELECT`, `WHERE`, `ORDER BY`, even inside aggregates.

CASE inside aggregates: conditional counting

```sql
SELECT
  COUNT(*)                                                AS total,
  SUM(CASE WHEN status = 'completed' THEN 1 ELSE 0 END) AS completed,
  SUM(CASE WHEN status = 'cancelled' THEN 1 ELSE 0 END) AS cancelled
FROM orders;
```

- ”Pivot”-style counting in one pass — a very common reporting pattern.
- Postgres also offers `COUNT(*) FILTER (WHERE status='completed')`.

Views — naming a query

```sql
CREATE VIEW cs_students AS
SELECT id, name, tot_cred
FROM   student
WHERE  dept_name = 'Comp. Sci.';

SELECT * FROM cs_students WHERE tot_cred > 50;
```

- A **view** is a stored query that behaves like a virtual table.
- Not stored data — it runs the underlying query each time it's used.

Why views?

- **Simplify** complex queries behind a friendly name.
- **Reuse** logic across many queries/apps.
- **Security** — expose only certain columns/rows (Lecture 11).
- **Stability** — apps depend on the view, not the raw schema.

We revisit updatable & materialized views in Lecture 11.

Summary — key takeaways

- **CTEs (`WITH`)** make complex queries readable and allow recursion.
- NULL ⇒ **three-valued logic**; test with `IS NULL`; comparisons give UNKNOWN.
- `COALESCE`/`NULLIF`/`CASE` compute values and tame NULLs.
- **Views** name queries for reuse and security.

Exercises (on `store.sql`)

1. Write a query using a CTE to find the top customer per country.
2. Show each customer's city, using 'Unknown' when NULL.
3. Count orders by status in a single row using `CASE`/`FILTER`.
4. Create a view `product_sales(prod_id, prod_name, units, revenue)` and query it.

Decision guide — NULLs & conditionals

- Test presence → `IS [NOT] NULL`.
- Provide a default → `COALESCE`.
- Avoid divide-by-zero → `/ NULLIF(x, 0)`.
- Branch a value → `CASE`.
- Conditional count → `SUM(CASE ...)` or `COUNT(*) FILTER (...)`.

Before next lecture

- Review views and CTEs.
- Optional reading: Silberschatz §3.6 (NULLs), §4.2 (views).

Next time — Lecture 9b: Window Functions.

Questions?