CSL303/MAL505 — Database Management Systems

Module I: Relational Foundations & SQL
Lecture 9: Remaining Subqueries

Amit Kumar Dhar

IIT Bhilai

Lecture 9: NULLs, CASE, Views & Window Functions

Module I — Relational Foundations & SQL

Three-valued logic, conditional expressions, views, and analytics

Agenda

- Remaining Subqueries: `ANY`/`ALL`, `EXISTS`, Correlated

ANY / SOME and ALL

```sql
-- price greater than SOME product in category 1
WHERE price > ANY (SELECT price FROM products WHERE cat_id = 1)

-- price greater than EVERY product in category 1
WHERE price > ALL (SELECT price FROM products WHERE cat_id = 1)
```

- `> ANY` = greater than the **minimum**; `> ALL` = greater than the **maximum**.
- `= ANY` is equivalent to `IN`. `<> ALL` is equivalent to `NOT IN`.

Quick Check: ANY/ALL

Question: Rewrite the condition `salary < ALL (SELECT salary FROM managers)` using an aggregate function.

Quick Check: ANY/ALL

Question: Rewrite the condition `salary < ALL (SELECT salary FROM managers)` using an aggregate function.

Answer:

```
WHERE salary < (SELECT MIN(salary) FROM managers)
```

EXISTS — does the subquery return anything?

```sql
SELECT c.name
FROM   customers c
WHERE  EXISTS (SELECT 1 FROM orders o WHERE o.cust_id = c.cust_id);
```

- `EXISTS` is **true** if the subquery yields ≥ 1 row (the columns don't matter, hence `SELECT 1`).
- Naturally a **semijoin**: "customers who have at least one order."

Quick Check: EXISTS

Question: Write a query using `EXISTS` to find products that have never been ordered.

Quick Check: EXISTS

Question: Write a query using `EXISTS` to find products that have never been ordered.

Answer:

```
SELECT p.prod_name
FROM products p
WHERE NOT EXISTS (
    SELECT 1 FROM order_items oi WHERE oi.prod_id = p.prod_id
);
```

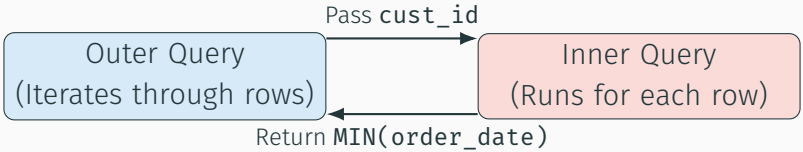
Correlated subqueries

- A **correlated** subquery references a column from the **outer** query.
- It is (conceptually) re-evaluated **for each outer row**.

```
SELECT o.order_id, o.cust_id
FROM   orders o
WHERE  o.order_date > (SELECT MIN(o2.order_date)
                       FROM   orders o2
                       WHERE  o2.cust_id = o.cust_id);
                       -- correlation on cust_id
```

- ”Orders that are not a customer’s first order.”

Execution Flow: Correlated Subquery



Correlated vs uncorrelated

	Uncorrelated	Correlated
References outer row?	No	Yes
Evaluated	once	per outer row (logically)
Example	`> (SELECT AVG(price)...)`	`WHERE o2.cust_id = o.`

- Optimizers often **rewrite** correlated subqueries into joins — but writing them clearly matters first.

NOT EXISTS — the anti-join

”Customers who have never ordered”:

```
SELECT c.name
FROM   customers c
WHERE  NOT EXISTS (SELECT 1 FROM orders o WHERE o.cust_id = c.cust_id);
```

- Robust against NULLs (unlike `NOT IN`).
- The idiomatic way to express ”has no matching rows.”

Division: the double NOT EXISTS

”Customers who have bought *every* product in category 1”:

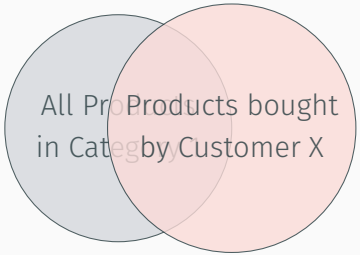
```
SELECT c.cust_id
FROM   customers c
WHERE  NOT EXISTS (                          -- there is no category-1 product..
    SELECT 1 FROM products p WHERE p.cat_id = 1
    AND NOT EXISTS (                         -- ...that this customer did NOT bu
        SELECT 1 FROM orders o JOIN order_items oi ON oi.order_id = o.order_id
        WHERE o.cust_id = c.cust_id AND oi.prod_id = p.prod_id));
```

- ”No required product is missing” = ”bought them all” — RA **division**.

Reading the double negation

- Outer `NOT EXISTS`: *there does not exist…*
- …a required product **p**…
- inner `NOT EXISTS`: *…that the customer did **not** buy.*
- ⇒ every required product **was** bought.
- Memorize the shape — it's the standard "for all" template in SQL.

Double NOT EXISTS Logic



Goal: Set of required products
is a subset of bought products.
Check: Is the set of required products that
are NOT in the bought products EMPTY?

Subqueries in FROM (derived tables)

- Query the result of a query. **Must** be aliased.

```sql
SELECT tier, AVG(revenue) AS avg_rev
FROM (
    SELECT o.cust_id,
           SUM(oi.quantity * oi.unit_price) AS revenue,
           CASE WHEN COUNT(*) > 5 THEN 'frequent' ELSE 'occasional' END AS tier
    FROM   orders o JOIN order_items oi ON oi.order_id = o.order_id
    GROUP  BY o.cust_id
) AS cust_summary
GROUP BY tier;
```

- Aggregating an aggregate → derived tables (or CTEs) are the tool.

Common Table Expressions (WITH)

- A CTE names a subquery at the top, then uses it like a table.

```
WITH cust_revenue AS (
    SELECT o.cust_id, SUM(oi.quantity * oi.unit_price) AS revenue
    FROM   orders o JOIN order_items oi ON oi.order_id = o.order_id
    GROUP  BY o.cust_id
)
SELECT c.name, r.revenue
FROM   cust_revenue r JOIN customers c ON c.cust_id = r.cust_id
WHERE  r.revenue > 20000;
```

Quick Check: WITH

Question: Rewrite the following using a CTE named `high_spenders`: `SELECT * FROM (SELECT cust_id FROM orders GROUP BY cust_id HAVING sum(total) > 1000) h;`

Quick Check: WITH

Question: Rewrite the following using a CTE named `high_spenders`: `SELECT * FROM (SELECT cust_id FROM orders GROUP BY cust_id HAVING sum(total) > 1000) h;`

Answer:

```sql
WITH high_spenders AS (
    SELECT cust_id FROM orders
    GROUP BY cust_id HAVING sum(total) > 1000
)
SELECT * FROM high_spenders;
```

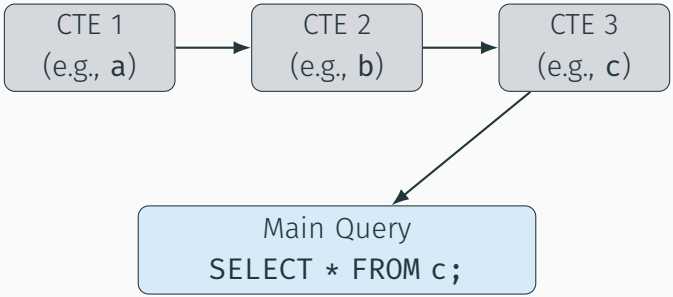
Why CTEs?

- **Readability** — name intermediate steps instead of deep nesting.
- **Reuse** — reference the same CTE multiple times in one query.
- **Chaining** — define several CTEs, each building on the last:

```
WITH a AS (...),
     b AS (SELECT ... FROM a ...),
     c AS (SELECT ... FROM b ...)
SELECT * FROM c;
```

- Both SQLite and PostgreSQL support CTEs.

CTE Data Flow



CTE vs derived table vs view

- **Derived table** — subquery in `FROM`, scoped to that query.
- **CTE** — named subquery, scoped to the statement, more readable.
- **View** — a *named, stored* query reusable across many statements (Lecture 11).
- Same idea, different lifetime/scope.

Recursive CTEs

- Handle **hierarchies / graphs**: org charts, category trees, prerequisites.

```
WITH RECURSIVE ancestors(course_id, prereq_id) AS (
    SELECT course_id, prereq_id FROM prereq         -- base case
    UNION
    SELECT a.course_id, p.prereq_id                 -- recursive step
    FROM   ancestors a JOIN prereq p ON p.course_id = a.prereq_id
)
SELECT * FROM ancestors;
```

- "All (transitive) prerequisites of each course."

Anatomy of a recursive CTE

1. **Base case** — the initial rows (`SELECT ... FROM prereq`).
2. **`UNION` / `UNION ALL`** — combine base with recursive results.
3. **Recursive step** — references the CTE itself, extends by one level.
4. Repeats until no new rows are produced.

- Use `UNION` (not `ALL`) to avoid infinite loops on cyclic data.

Rewriting subquery ⇄ join

- Many `IN`/`EXISTS` subqueries can become joins, and vice versa.

```sql
-- subquery form
SELECT name FROM customers
WHERE cust_id IN (SELECT cust_id FROM orders WHERE status = 'completed');

-- join form (add DISTINCT to avoid duplicates)
SELECT DISTINCT c.name FROM customers c
JOIN orders o ON o.cust_id = c.cust_id WHERE o.status = 'completed';
```

- Choose whichever reads clearer; the optimizer often produces the same plan.

Subquery vs join: watch duplicates

- `IN`/`EXISTS` return each outer row **once**.
- A join to a one-to-many table returns it **many** times → add `DISTINCT`.
- If you only need "does a match exist," **`EXISTS` is cleaner** and avoids the duplicate problem entirely.

Scalar subquery per row (correlated in SELECT)

”Each product with how many orders included it”:

```
SELECT p.prod_name,
       (SELECT COUNT(*) FROM order_items oi WHERE oi.prod_id = p.prod_id)
         AS times_ordered
FROM   products p;
```

- A correlated scalar subquery in the `SELECT` list — one value per outer row.
- Often rewritable as a `LEFT JOIN ... GROUP BY` (usually faster).

Subqueries in HAVING

- The subquery can supply a threshold for group filtering:

```
SELECT cust_id, SUM(oi.quantity*oi.unit_price) AS spent
FROM   orders o JOIN order_items oi ON oi.order_id = o.order_id
GROUP  BY cust_id
HAVING SUM(oi.quantity*oi.unit_price) >
       (SELECT AVG(total) FROM (
           SELECT SUM(quantity*unit_price) AS total
           FROM order_items GROUP BY order_id) t);
```

- ”Customers who spent more than the average order value.”

Multiple-column (row) subqueries

- Compare a **tuple** to a subquery result:

```
SELECT * FROM order_items oi
WHERE (oi.prod_id, oi.unit_price) IN
      (SELECT prod_id, price FROM products);   -- items sold at list price
```

- Row constructors let `IN`/comparison work over several columns at once (supported by PostgreSQL; SQLite supports row-value `IN` since 3.15).

Correlated UPDATE/DELETE (preview)

- Subqueries aren't just for `SELECT` — they drive updates too:

```
DELETE FROM products p
WHERE NOT EXISTS (
    SELECT 1 FROM order_items oi WHERE oi.prod_id = p.prod_id
);
-- remove products that were never ordered
```

- Full DML with subqueries is Lecture 10; note the pattern now.

Summary — key takeaways

- `> ANY` means greater than the minimum; `> ALL` means greater than the maximum.
- `EXISTS` tests whether a subquery returns at least one row, and is often faster because it short-circuits.
- **Correlated subqueries** reference columns from the outer query and logically evaluate row-by-row.
- Be careful with the `NOT IN` + `NULL` trap; use `NOT EXISTS` instead.

Exercises (on `store.sql`)

1. Find customers who have never placed an order using `NOT EXISTS`.
2. Find products whose price is greater than the average price in their own category.
3. Rewrite a `NOT IN` query safely when the subquery might return `NULL`.

Quick reference — subqueries

- **Scalar:** Returns 1 row, 1 column. Can be used anywhere a value is expected.
- **Correlated:** References the outer query. Evaluates repeatedly.
- **EXISTS:** Returns TRUE if ≥ 1 row. Ignores the `SELECT` list (`SELECT 1`).
- **ANY / ALL:** Compares a single value to a set of values.

Before next lecture

- Review how `FROM` clauses work. We will be placing queries inside `FROM` next time!
- Try writing the same queries using both Joins and Subqueries to understand the difference.

Next time — Lecture 9a: Derived Tables, CTEs, Analytics & Views.

Questions?