CSL303/MAL505 — Database Management Systems

Module I: Relational Foundations & SQL
Lecture 8: Subqueries & CTEs

Amit Kumar Dhar

IIT Bhilai

Why subqueries?

- Some questions need the **result of one query inside another**:
  - ”products priced above the **average** price,”
  - ”customers who bought **every** product in a category,”
  - ”orders larger than that customer’s **own** average.”
- A **subquery** is a `SELECT` nested inside another statement.
- **CTEs** (`WITH`) let us name and reuse subqueries for readability.

Agenda

- Scalar subqueries
- Subqueries in `WHERE`: `IN` and the NULL trap

The FILTER clause (conditional aggregates)

```sql
SELECT cust_id,
       COUNT(*)                                    AS all_orders,
       COUNT(*) FILTER (WHERE status='completed') AS completed,
       COUNT(*) FILTER (WHERE status='cancelled') AS cancelled
FROM   orders
GROUP  BY cust_id;
```

- `FILTER (WHERE ...)` aggregates only matching rows (PostgreSQL).
- Portable equivalent: `SUM(CASE WHEN status='completed' THEN 1 ELSE 0 END)`.

Empty groups don’t exist

- `GROUP BY` produces a group **only if at least one row falls into it**.
- ”Categories with zero products” won’t appear from `GROUP BY products.cat_id`.
- To include them, start from `categories` and **LEFT JOIN** to products, then group — the outer join manufactures the empty-group rows.

Combining WHERE, GROUP BY, HAVING, ORDER BY

”Countries (excluding the UK) with ≥ 2 customers who joined since 2022, most customers first”:

```sql
SELECT country, COUNT(*) AS n
FROM   customers
WHERE  country <> 'UK' AND join_date >= '2022-01-01'
GROUP  BY country
HAVING COUNT(*) >= 2
ORDER  BY n DESC;
```

- All the clauses in one query — note each does a distinct job.

Aggregates can nest with subqueries

- ”Category whose average price is the highest”:

```
SELECT cat_id, AVG(price) AS a
FROM   products
GROUP  BY cat_id
HAVING AVG(price) = (SELECT MAX(cat_avg)
                     FROM (SELECT AVG(price) AS cat_avg
                           FROM products GROUP BY cat_id) t);
```

- ”Aggregate of an aggregate” needs a subquery/CTE — a bridge to Lecture 8.

Statistical aggregates (bonus)

- Beyond the big five, engines offer more:
  - `STDDEV`, `VARIANCE` (Postgres),
  - `TOTAL` (SQLite's NULL-safe SUM that returns 0.0),
  - `MODE()`, percentile functions (Postgres ordered-set aggregates).
  - Know they exist; reach for them when a report needs them.

A grouped report, end to end

”Per category: number of products, average price, total units sold, revenue.”

```sql
SELECT c.cat_name,
       COUNT(DISTINCT p.prod_id)            AS n_products,
       ROUND(AVG(p.price), 2)               AS avg_price,
       COALESCE(SUM(oi.quantity), 0)        AS units_sold,
       COALESCE(SUM(oi.quantity*oi.unit_price), 0) AS revenue
FROM   categories c
LEFT JOIN products p     ON p.cat_id = c.cat_id
LEFT JOIN order_items oi ON oi.prod_id = p.prod_id
GROUP  BY c.cat_name
ORDER  BY revenue DESC;
```

Common mistakes

- Selecting a non-grouped, non-aggregated column (silent bug in SQLite).
- Using `WHERE` where you needed `HAVING` (or vice versa).
- `COUNT(*)` vs `COUNT(col)` with outer joins.
- Double-counting after a one-to-many join.
- Referencing a `SELECT` alias in `WHERE`/`GROUP BY`.

Percent-of-total pattern

- Show each group's share of the whole:

```
SELECT cat_id,
       SUM(stock) AS units,
       ROUND(100.0 * SUM(stock) / (SELECT SUM(stock) FROM products), 1) AS pct
FROM   products
GROUP  BY cat_id;
```

- The subquery computes the grand total once; each group divides by it.
- (Window functions offer `SUM(...) OVER ()` for the same effect — Lecture 9.)

Grouping by a boolean condition

```sql
SELECT price >= 2000 AS is_premium, COUNT(*) AS n, AVG(stock) AS avg_stock
FROM   products
GROUP  BY price >= 2000;
```

- Group on a **predicate** to split rows into two buckets (true/false).
- A quick way to compare two segments without a full CASE.

Sanity-checking your aggregates

- Always cross-check: does **SUM** over all groups equal the ungrouped **SUM**?
- Row counts: `SELECT COUNT(*)` before grouping vs `SUM(group_counts)` after.
- If a **SUM** looks too big after a join, suspect **row multiplication**.
- Spot-check one group by hand on a few rows.

Aggregates are where silent errors hide — verify, don't trust.

Three places subqueries appear

1. In `SELECT` — a *scalar* subquery returning one value.
2. In `WHERE/HAVING` — with `IN`, `EXISTS`, `ANY`, `ALL`, or comparison.
3. In `FROM` — a *derived table* (a subquery you query further).

Scalar subquery

- Returns **exactly one row and one column** → usable like a single value.

```
SELECT prod_name, price,
       (SELECT AVG(price) FROM products) AS avg_price
FROM   products;
```

- Here the subquery computes the overall average once, shown on every row.
- If a scalar subquery returns >1 row → runtime error.

Scalar subquery in WHERE

”Products priced above the average price”:

```
SELECT prod_name, price
FROM   products
WHERE  price > (SELECT AVG(price) FROM products);
```

- The inner query yields one number; the outer compares each row to it.
- This is impossible with a plain `WHERE` (can’t put an aggregate in `WHERE`).

IN — membership in a subquery result

”Products in the ’Electronics’ or ’Books’ categories”:

```
SELECT prod_name
FROM   products
WHERE  cat_id IN (SELECT cat_id FROM categories
                  WHERE cat_name IN ('Electronics','Books'));
```

- `IN` tests membership in the set the subquery returns.
- `NOT IN` for exclusion — but beware NULLs (next slide).

Quick Check: IN

Question: How would you find all customers whose IDs are in the list of customers that made an order today?

Quick Check: IN

Question: How would you find all customers whose IDs are in the list of customers that made an order today?

Answer:

```
SELECT name
FROM customers
WHERE cust_id IN (
    SELECT cust_id FROM orders WHERE order_date = CURRENT_DATE
);
```

The NOT IN + NULL trap

```sql
-- if the subquery returns any NULL, NOT IN yields NO rows!
SELECT name FROM customers
WHERE  cust_id NOT IN (SELECT cust_id FROM orders);
-- risky if cust_id can be NULL
```

- `x NOT IN (a, b, NULL)` evaluates to **unknown**, never true.
- Safer: use **`NOT EXISTS`** (immune to this) or filter NULLs in the subquery.

Summary — key takeaways

- Subqueries appear in **`SELECT`, `WHERE`/`HAVING`,** and **`FROM`**.
- `EXISTS`/`NOT EXISTS` express semi/anti-joins robustly (NULL-safe).
- **Correlated** subqueries reference the outer row (logically per-row).
- **CTEs (`WITH`)** name and chain subqueries; **recursive CTEs** handle hierarchies.
- "For all" queries (division) $\rightarrow$ the **double `NOT EXISTS`** idiom.

Exercises (on `store.sql`)

1. Products priced above their **category's** average price (correlated).
2. Customers who ordered in 2023 but **not** in 2022 (`EXISTS`/`NOT EXISTS`).
3. Using a CTE, list the top-spending customer per country.
4. Customers who bought **every** product in the 'Books' category (division).
5. Recursive CTE: all transitive prerequisites of `CS-347` (on `university.sql`).

Quick reference — subquery syntax

```
-- scalar
WHERE x > (SELECT AVG(x) FROM t)
-- membership
WHERE x IN (SELECT x FROM t)        WHERE x NOT IN (...)   -- NULL-risky
-- existence
WHERE EXISTS (SELECT 1 FROM t WHERE ...)   WHERE NOT EXISTS (...)
-- quantified
WHERE x > ALL (...)                 WHERE x > ANY (...)
-- derived table
FROM (SELECT ...) AS d
-- CTE
WITH c AS (SELECT ...) SELECT ... FROM c
-- recursive
WITH RECURSIVE r AS (base UNION step) SELECT * FROM r
```

Before next lecture

- Try each exercise; compare a subquery form with its join rewrite.
- Optional reading: Silberschatz §3.8 (nested subqueries), §5.4 (recursion).

Next time — Lecture 9: NULLs & three-valued logic, CASE, **views**, and **window functions**.

Questions?