Advanced Query Techniques

DATA 503: Fundamentals of Data Engineering

Lucas P. Cordova, Ph.D.

2026-04-08

A concise version of advanced SQL patterns: restoring the practice database, subqueries in WHERE and FROM, IN and EXISTS, LATERAL for top-N per group, CTEs for readability, a short crosstab example, and CASE for classification. Expect checkpoints with hints, then solutions on the following slide. Based on Lecture 11-1 and Chapter 13 of Practical SQL, 2nd Edition.

Table of contents

1 Learning objectives

1.1 What you should be able to do

1. Use scalar and derived-table subqueries for filtering and reshaping aggregates
2. Choose `IN` vs `EXISTS` / `NOT EXISTS` for membership tests and anti-joins
3. Write `LATERAL` joins for per-row computations and top-N patterns
4. Refactor multi-step logic with CTEs (`WITH`) and optional `CASE` classification

. . .

Course connection: These patterns show up in analytical SQL, incremental pipelines, and any time you need correct, readable transformations over relational data.

2 Restore the practice database

2.1 Create a Database

Create a database called class_dbs using the `createdb` command in a terminal OR using any method you usually use to create a new database.

```
createdb -h localhost -U postgres class_dbs
```

2.2 Download the files from Canvas

You will get **one** of these (not both required):

File	Format	Tool
`class_dbs.sql`	Plain SQL script	`psql` with `-f`
`class_dbs.dump`	Custom `pg_dump -Fc` archive	`pg_restore`

Save the file somewhere easy to find. The examples below assume it landed in your **Downloads** folder.

2.3 Local PostgreSQL: plain SQL (`psql`)

Open a terminal, go to the folder that contains `class_dbs.sql`, then run `psql` against the `class_dbs` database you created above.

```
cd ~/Downloads
psql -h localhost -U postgres -d class_dbs -f class_dbs.sql
```

2.4 Local PostgreSQL: custom archive (`pg_restore`)

The `class_dbs` database must exist before restore (same as the SQL path above).

```
cd ~/Downloads
pg_restore -h localhost -U postgres -d class_dbs --verbose class_dbs.dump
```

2.5 Railway: same files, database `railway`

Replace host and port with the values from your Railway Postgres service (Variables / Connect). The database name is often `railway`.

Plain SQL:

```
cd ~/Downloads
psql "postgresql://postgres:YOUR_PASSWORD@YOUR_HOST:YOUR_PORT/railway" -f class_dbs.sql
```

Custom dump:

```
cd ~/Downloads
pg_restore "postgresql://postgres:YOUR_PASSWORD@YOUR_HOST:YOUR_PORT/railway" --verbose class_
```

3 Subqueries: WHERE and FROM

3.1 Scalar subquery in `WHERE`

A **scalar** subquery returns one value. The database can evaluate it once, then use it like a constant in the outer query.

Run this. Counties at or above the national 90th percentile for `pop_est_2019`:

```sql
SELECT county_name, state_name, pop_est_2019
FROM us_counties_pop_est_2019
WHERE pop_est_2019 >= (
    SELECT percentile_cont(0.9) WITHIN GROUP (ORDER BY pop_est_2019)
    FROM us_counties_pop_est_2019
)
ORDER BY pop_est_2019 DESC;
```

This subquery is **uncorrelated**: it does not reference the outer row. It typically runs once, then the outer query filters.

3.2 Derived table in `FROM`

A subquery in `FROM` builds a **derived table**. PostgreSQL requires an alias (`AS ...`).

```sql
SELECT round(calcs.avg_pop, 0) AS avg_pop,
       calcs.median_pop,
       round(calcs.avg_pop - calcs.median_pop, 0) AS mean_minus_median
FROM (
    SELECT avg(pop_est_2019) AS avg_pop,
           percentile_cont(0.5)
               WITHIN GROUP (ORDER BY pop_est_2019)::numeric AS median_pop
    FROM us_counties_pop_est_2019
) AS calcs;
```

Use this when you need to aggregate first, then treat the result as a table you can select from or join.

3.3 Checkpoint: above-average counties

Task: Return every county whose `pop_est_2019` is **strictly greater** than the national average.

Required columns: `county_name`, `state_name`, `pop_est_2019`

Requirements:

- Use a **scalar subquery** in `WHERE` that computes `avg(pop_est_2019)` over all counties
- Order by `pop_est_2019` descending

Expected shape (first rows, illustrative):

county_name	state_name	pop_est_2019
Los Angeles County	California	10039107
Cook County	Illinois	5150233
Harris County	Texas	4713320

Hint: Compare `pop_est_2019` to `(SELECT avg(pop_est_2019) FROM us_counties_pop_est_2019)`.

3.4 Solution: above-average counties

```sql
SELECT county_name, state_name, pop_est_2019
FROM us_counties_pop_est_2019
WHERE pop_est_2019 > (
    SELECT avg(pop_est_2019)
    FROM us_counties_pop_est_2019
)
ORDER BY pop_est_2019 DESC;
```

Why fewer than half the rows?

4 `IN` and `EXISTS`

4.1 `IN (subquery)`

`IN` tests membership against the set returned by the subquery.

```sql
SELECT first_name, last_name
FROM employees
WHERE emp_id IN (
    SELECT id FROM retirees
)
ORDER BY emp_id;
```

Readable when the subquery is small and uncorrelated.

4.2 `EXISTS` and `NOT EXISTS`

`EXISTS` returns true if the subquery returns **at least one row**. The subquery can reference outer columns; that makes it **correlated** (conceptually: evaluated per outer row, though the planner may rewrite).

```sql
SELECT first_name, last_name
FROM employees AS e
WHERE EXISTS (
    SELECT 1
    FROM retirees AS r
    WHERE r.id = e.emp_id
);
```

`NOT EXISTS` is the usual pattern for anti-joins: rows in A with no match in B. Prefer `NOT EXISTS` over `NOT IN` when the subquery can produce NULLs, because `NOT IN` with NULLs in the list behaves badly.

4.3 Checkpoint: active employees

Task: List employees who are **not** in `retirees`.

Required columns: `first_name`, `last_name`

Requirements: Use `NOT EXISTS` with a correlated subquery matching `retirees.id` to `employees.emp_id`. Order by `last_name`, `first_name`

Expected shape:

first_name	last_name
Samuel	Cole
Arthur	Pappas
…	…

. . .

Hint: `SELECT 1` inside `EXISTS` is enough; you only care whether a row appears.

4.4 Solution: active employees

```sql
SELECT first_name, last_name
FROM employees AS e
WHERE NOT EXISTS (
    SELECT 1
    FROM retirees AS r
    WHERE r.id = e.emp_id
)
ORDER BY last_name, first_name;
```

5 LATERAL joins

5.1 Top-N per group

LATERAL lets a subquery in FROM reference columns from earlier tables. Classic use: **for each parent row**, run a query and keep a few rows.

. . .

For each teacher, return their two most recent lab accesses.

```sql
SELECT t.first_name, t.last_name, a.access_time, a.lab_name
FROM teachers AS t
LEFT JOIN LATERAL (
    SELECT *
    FROM teachers_lab_access
    WHERE teacher_id = t.id
    ORDER BY access_time DESC
    LIMIT 2
) AS a ON true
ORDER BY t.id;
```

. . .

LEFT JOIN LATERAL keeps teachers with zero visits (NULLs in a.*). ON true satisfies join syntax; filtering is inside the lateral subquery.

5.2 Checkpoint: one visit per teacher

Task: Same as the demo, but return only each teacher's **single most recent** lab access.

. . .

Required columns:

- `first_name`
- `last_name`
- `access_time`
- `lab_name`

. . .

Requirements:

- `LEFT JOIN LATERAL` with `ORDER BY access_time DESC` and `LIMIT 1`

. . .

Expected shape (illustrative):

first_name	last_name	access_time	lab_name
Janet	Smith	2022-12-21 …	Chemistry A
Lee	Reynolds	…	…

. . .

Hint: Change one number from the demo query.

5.3 Solution: one visit per teacher

```sql
SELECT t.first_name, t.last_name, a.access_time, a.lab_name
FROM teachers AS t
LEFT JOIN LATERAL (
    SELECT *
    FROM teachers_lab_access
    WHERE teacher_id = t.id
    ORDER BY access_time DESC
    LIMIT 1
) AS a ON true
ORDER BY t.id;
```

6 Review: Common Table Expressions (`WITH`)

6.1 Why CTEs

A CTE names an intermediate result. Use them when you:

. . .

- chain several logical steps
- reuse the same expression instead of repeating a subquery
- want readers (including future you) to follow the story top to bottom

. . .

```sql
WITH big_counties AS (
    SELECT county_name, state_name, pop_est_2019
    FROM us_counties_pop_est_2019
    WHERE pop_est_2019 >= 100000
)
SELECT state_name, count(*) AS num_big_counties
FROM big_counties
GROUP BY state_name
ORDER BY num_big_counties DESC;
```

6.2 Checkpoint: state totals with two CTEs

Task: Build a state-level summary (table: `us_counties_pop_est_2019`)

. . .

Required columns:

- `state_name`
- `total_pop` (sum of `pop_est_2019`)
- `num_counties` (count of rows per state)
- `avg_pop_per_county` (`total_pop` / `num_counties`, rounded to whole numbers is fine)

. . .

Requirements:

- **CTE 1:** aggregate population by `state_name`
- **CTE 2:** aggregate county counts by `state_name`
- Join the two CTEs on `state_name`
- Order by `avg_pop_per_county` descending

. . .

Expected shape (first rows):

state_name	total_pop	num_counties	avg_pop_per_county
California	…	…	…
Texas	…	…	…

. . .

Hint: Same `GROUP BY state_name` in both CTEs; join keys match exactly.

6.3 Solution: state totals with two CTEs

```sql
WITH state_pop AS (
    SELECT state_name, sum(pop_est_2019) AS total_pop
    FROM us_counties_pop_est_2019
    GROUP BY state_name
),
state_counties AS (
    SELECT state_name, count(*) AS num_counties
    FROM us_counties_pop_est_2019
    GROUP BY state_name
)
SELECT sp.state_name,
       sp.total_pop,
       sc.num_counties,
       round(sp.total_pop::numeric / sc.num_counties, 0) AS avg_pop_per_county
FROM state_pop AS sp
JOIN state_counties AS sc USING (state_name)
ORDER BY avg_pop_per_county DESC;
```

7 Crosstab

7.1 Enable and pivot

Reporting often needs **long to wide** layout. PostgreSQL provides `crosstab()` in the `tablefunc` extension.

. . .

```sql
CREATE EXTENSION IF NOT EXISTS tablefunc;

SELECT *
FROM crosstab(
    $$
    SELECT office, flavor, count(*) AS response_count
    FROM ice_cream_survey
    GROUP BY office, flavor
    ORDER BY office
    $$,
    $$ SELECT flavor
       FROM ice_cream_survey
       GROUP BY flavor
       ORDER BY flavor
    $$
) AS ct(
    office text,
    chocolate bigint,
    strawberry bigint,
    vanilla bigint
);
```

. . .

Contract: the first query must expose **three** columns: row id, category, value. The second query lists category labels. The outer `AS (...)` names output columns in that order.

8 `CASE` and classification

8.1 Pattern

`CASE` evaluates **first match wins**. Put the most specific conditions first when ranges overlap.

```sql
SELECT max_temp,
       CASE
           WHEN max_temp >= 90 THEN 'Hot'
           WHEN max_temp >= 70 THEN 'Warm'
           WHEN max_temp >= 50 THEN 'Pleasant'
           ELSE 'Cool or missing'
       END AS temp_band
FROM temperature_readings
LIMIT 5;
```

8.2 Checkpoint: population tiers with a CTE

Task: Classify each county, then summarize. (table: `us_counties_pop_est_2019`)

Required columns (final result):

- `tier`
- `num_counties`
- `total_pop`
- `avg_county_pop`

Requirements:

- Use a **CTE** that adds `tier` with `CASE`
- Outer query: `GROUP BY tier` with aggregates

Tier rules on `pop_est_2019`:

- `Metro`: `>= 500000`
- `Urban`: 100000 to 499999
- `Suburban`: 50000 to 99999
- `Rural`: below 50000

Expected shape:

tier	num_counties	total_pop	avg_county_pop
Rural	...	...	...
Metro	...	...	...

Hint: `CASE` should mirror the order above so a county falls into exactly one tier.

8.3 Solution: population tiers

```sql
WITH county_tiers AS (
    SELECT county_name,
           state_name,
           pop_est_2019,
           CASE
               WHEN pop_est_2019 >= 500000 THEN 'Metro'
               WHEN pop_est_2019 >= 100000 THEN 'Urban'
               WHEN pop_est_2019 >= 50000 THEN 'Suburban'
               ELSE 'Rural'
           END AS tier
```

```
    FROM us_counties_pop_est_2019
)
SELECT tier,
       count(*) AS num_counties,
       sum(pop_est_2019) AS total_pop,
       round(avg(pop_est_2019), 0) AS avg_county_pop
FROM county_tiers
GROUP BY tier
ORDER BY total_pop DESC;
```

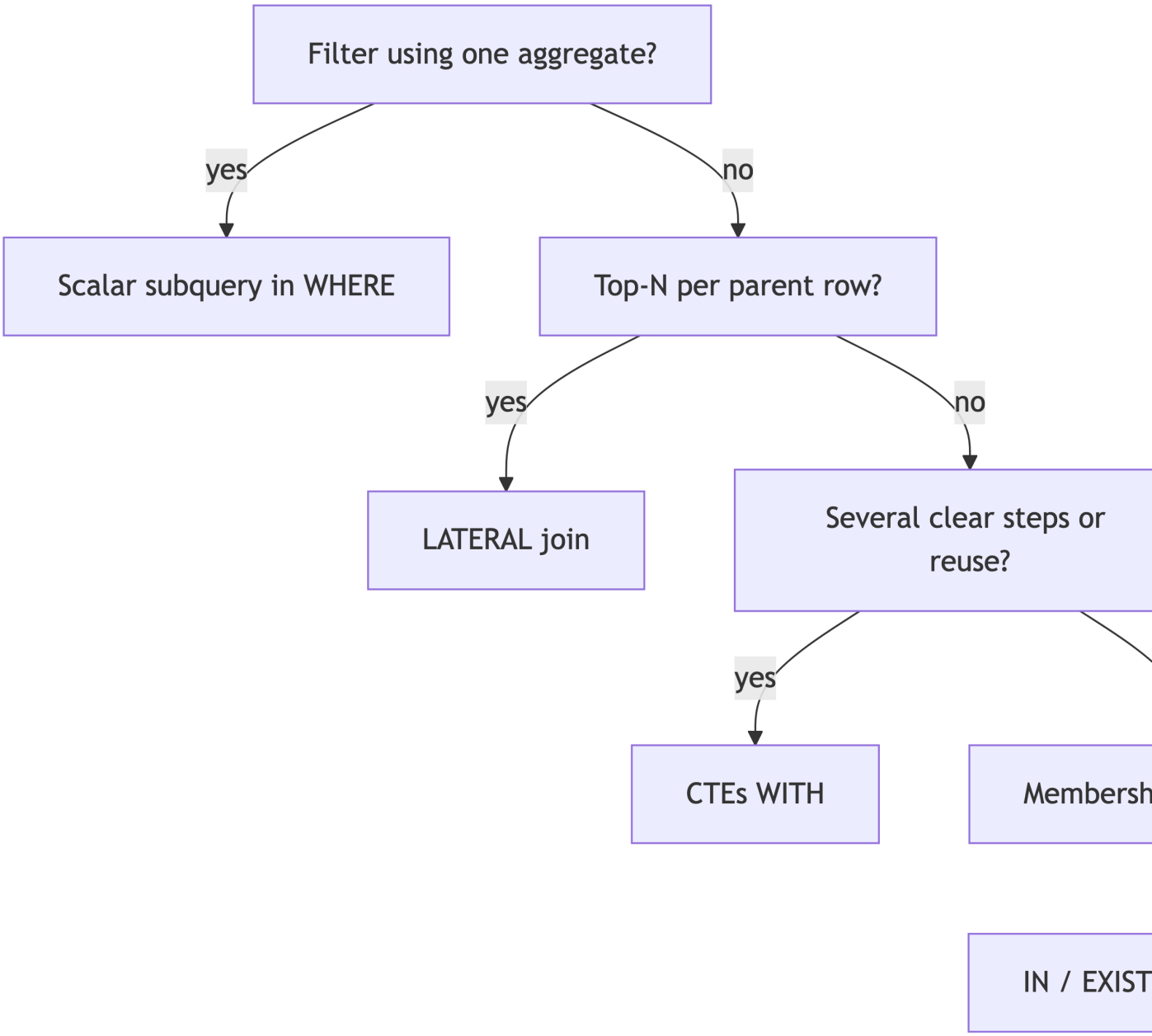
Typical story: many counties are rural, but a small number of metro counties hold a large share of total population.

9 Summary

9.1 Techniques in one place

Technique	Use it when
Scalar subquery in `WHERE`	Compare rows to one computed threshold
Derived table in `FROM`	Aggregate, then select or join the aggregate
`IN`	Simple membership against a set
`EXISTS` / `NOT EXISTS`	Correlated checks; anti-joins; nullable-safe negation
`LATERAL`	Per-row subquery, top-N per group
CTE (`WITH`)	Multiple named steps, less repetition
`crosstab()`	Pivot reporting shapes
`CASE`	Bucket values before or during aggregation

9.2 Choosing a pattern



9.3 References

1. DeBarros, A. (2022). *Practical SQL* (2nd ed.). No Starch Press. Chapter 13.

2. PostgreSQL: WITH Queries (CTEs)
3. PostgreSQL: Subquery Expressions
4. PostgreSQL: LATERAL
5. PostgreSQL: tablefunc