Lecture 10-3: Advanced Query Techniques

DATA 503: Fundamentals of Data Engineering

Lucas P. Cordova, Ph.D.

2026-03-16

This lecture covers advanced SQL query techniques: subqueries in WHERE, FROM, and SELECT clauses, correlated subqueries, EXISTS, LATERAL joins, Common Table Expressions, cross tabulations, and CASE expressions. We apply these patterns to Census, survey, and temperature data. Based on Chapter 13 of Practical SQL, 2nd Edition.

Table of contents

1 Part 1: Setting Up

1.1 Database Setup

1.2 Create and Populate the Database

Create a fresh database for this lecture:

```sql
CREATE DATABASE advanced_queries;
```

Connect to it, then run the setup script provided: [**advanced_queries.sql**]

The script creates and loads the following tables:

Table	Rows	Description
`us_counties_pop_est_2019`	3,142	County population estimates
`cbp_naics_72_establishments`	2,764	Food/accommodation businesses per county
`employees`	6	Small HR demo table
`teachers`	6	Small demo table (from Ch 7)
`ice_cream_survey`	200	Office ice cream flavor preferences
`temperature_readings`	730	Two years of daily temps at two stations

All CSVs are in the `Chapter_13/` folder alongside these slides. Download them before class.

1.3 Verify Your Setup

```sql
SELECT count(*) FROM us_counties_pop_est_2019;    -- 3,142
SELECT count(*) FROM cbp_naics_72_establishments;  -- 2,764
SELECT count(*) FROM employees;                    -- 6
SELECT count(*) FROM ice_cream_survey;             -- 200
SELECT count(*) FROM temperature_readings;         -- 730
```

If any count is wrong, re-run the setup. We need all of these today.

2 Part 2: Subqueries

A query inside a query. It's queries all the way down.

2.1 Subqueries in WHERE

2.2 Scalar Subquery in WHERE

The most common pattern: use a subquery to compute a threshold, then filter on it.

```sql
SELECT county_name,
       state_name,
       pop_est_2019
FROM us_counties_pop_est_2019
WHERE pop_est_2019 >= (
    SELECT percentile_cont(.9) WITHIN GROUP (ORDER BY pop_est_2019)
    FROM us_counties_pop_est_2019
)
ORDER BY pop_est_2019 DESC;
```

Run this. The inner query computes the 90th percentile of county populations. The outer query returns only counties at or above that threshold.

The subquery executes first, returns a single value, and the outer query uses it like a constant. This is a **scalar subquery** – it returns exactly one value.

2.3 Subquery with DELETE

Subqueries work with DML too. Let's create a copy and prune it:

```sql
CREATE TABLE us_counties_2019_top10 AS
SELECT * FROM us_counties_pop_est_2019;

DELETE FROM us_counties_2019_top10
WHERE pop_est_2019 < (
    SELECT percentile_cont(.9) WITHIN GROUP (ORDER BY pop_est_2019)
    FROM us_counties_2019_top10
);

SELECT count(*) FROM us_counties_2019_top10;
```

Run this. You started with 3,142 counties and now have ~315 (the top 10%).

In a pipeline context: this is how you build filtered summary tables. Copy the source, delete what you don't need, keep the rest for downstream consumers.

2.4 Quick Quiz: Subquery Execution

When does the subquery in a WHERE clause execute?

```
A. Once before the outer query starts
B. Once per row of the outer query
C. After the outer query finishes
D. It depends
```

. . .

A (for scalar subqueries). The database computes the inner value once, then scans the outer table. Correlated subqueries (coming up) are different – they run once *per row*.

2.5 Try It: Above-Average Counties

Write a query that returns all counties where `pop_est_2019` is above the national average. Show county name, state, and population. How many are there?

```sql
-- Your query here
```

. . .

```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;
```

Far fewer than half! The mean is pulled up by massive counties (LA, Cook, Harris), so most counties are below average. Median vs. mean matters.

2.6 Subqueries in FROM (Derived Tables)

2.7 Using a Subquery as a Table

A subquery in the `FROM` clause creates a **derived table** – a temporary result set you can query like a regular table:

```sql
SELECT round(calcs.average, 0) AS average,
       calcs.median,
       round(calcs.average - calcs.median, 0) AS median_average_diff
FROM (
    SELECT avg(pop_est_2019) AS average,
           percentile_cont(.5)
               WITHIN GROUP (ORDER BY pop_est_2019)::numeric AS median
    FROM us_counties_pop_est_2019
) AS calcs;
```

Run this. The inner query computes two aggregates. The outer query calculates the difference. The `AS calcs` alias is required – PostgreSQL demands that derived tables have names.

2.8 Joining Two Derived Tables

This is powerful for combining aggregations from different tables:

```sql
SELECT census.state_name AS st,
       census.pop_est_2018,
       est.establishment_count,
       round((est.establishment_count / census.pop_est_2018::numeric) * 1000, 1)
           AS estabs_per_thousand
FROM
    (SELECT st, sum(establishments) AS establishment_count
     FROM cbp_naics_72_establishments
     GROUP BY st) AS est
JOIN
    (SELECT state_name, sum(pop_est_2018) AS pop_est_2018
     FROM us_counties_pop_est_2019
     GROUP BY state_name) AS census
ON est.st = census.state_name
ORDER BY estabs_per_thousand DESC;
```

Each derived table aggregates to the state level separately, then they join on state name. This is a common ETL pattern: aggregate separately, join at the grain you need.

2.9 Quick Quiz: Derived Table Rules

What happens if you forget the `AS` alias on a derived table?

```
A. PostgreSQL infers a name automatically
B. You get a syntax error
C. It works but you can't reference the columns
D. The query runs but returns wrong results
```

. . .

B. PostgreSQL requires every derived table to have an alias. `AS calcs`, `AS est`, `AS census` – pick a name. Without it, the parser rejects the query.

2.10 Subqueries in SELECT

2.11 Column-Level Subqueries

You can put a subquery directly in the SELECT list to add a computed column:

```sql
SELECT county_name,
       state_name AS st,
       pop_est_2019,
       (SELECT percentile_cont(.5) WITHIN GROUP (ORDER BY pop_est_2019)
        FROM us_counties_pop_est_2019) AS us_median
FROM us_counties_pop_est_2019;
```

Every row gets the same median value alongside its own population. This lets you compare each county to the national benchmark without a JOIN.

2.12 Subquery in a Calculation

Take it further – compute the difference from the median inline:

```sql
SELECT county_name,
       state_name AS st,
       pop_est_2019,
       pop_est_2019 - (SELECT percentile_cont(.5) WITHIN GROUP (ORDER BY pop_est_2019)
                       FROM us_counties_pop_est_2019) AS diff_from_median
FROM us_counties_pop_est_2019
WHERE (pop_est_2019 - (SELECT percentile_cont(.5) WITHIN GROUP (ORDER BY pop_est_2019)
                       FROM us_counties_pop_est_2019))
       BETWEEN -1000 AND 1000;
```

This finds counties whose population is within 1,000 of the national median. Notice the subquery appears twice – once in SELECT, once in WHERE. That redundancy is why CTEs exist (we'll get there).

2.13 Try It: Every County vs. Its State Average

Write a query that shows each county's population alongside its state's average population (using a correlated subquery in the SELECT list). Show county_name, state_name, pop_est_2019, and state_avg.

Hint: the subquery references the outer table's `state_name`.

```
-- Your query here
```

. . .

```
SELECT county_name,
       state_name,
       pop_est_2019,
       (SELECT round(avg(pop_est_2019), 0)
        FROM us_counties_pop_est_2019 AS inner_t
        WHERE inner_t.state_name = outer_t.state_name) AS state_avg
FROM us_counties_pop_est_2019 AS outer_t
ORDER BY state_name, pop_est_2019 DESC;
```

This is a **correlated subquery** – it runs once per row of the outer query because it references `outer_t.state_name`. More expensive than scalar subqueries, but very expressive.

3 Part 3: EXISTS and IN with Subqueries

Checking membership across tables – the SQL equivalent of "are you on the list?"

3.1 EXISTS, IN, and Correlated Subqueries

3.2 Setup: Employees and Retirees

```
CREATE TABLE retirees (
    id int,
    first_name text,
    last_name text
);

INSERT INTO retirees
VALUES (2, 'Janet', 'King'),
       (4, 'Michael', 'Taylor');
```

Now we have two employees who retired. Let's find them – and find who's still active.

3.3 Using IN with a Subquery

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

`IN` checks if `emp_id` matches any value in the subquery result. Simple and readable.

3.4 Using EXISTS (Correlated)

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

`EXISTS` returns TRUE if the subquery finds at least one row. The subquery references `employees.emp_id` from the outer query – making it correlated.

For small tables, `IN` and `EXISTS` perform identically. For large tables, `EXISTS` can be faster because it stops scanning as soon as it finds one match.

3.5 NOT EXISTS: The Anti-Join

```sql
SELECT first_name, last_name
FROM employees
WHERE NOT EXISTS (
    SELECT id
    FROM retirees
    WHERE id = employees.emp_id
);
```

Run this. Returns employees who are NOT retirees. This is an **anti-join** – one of the most useful patterns in data engineering. "Give me everything in table A that has no match in table B."

Pipeline use case: find new records that haven't been processed yet. `WHERE NOT EXISTS (SELECT 1 FROM processed WHERE processed.id = raw.id)`.

3.6 Quick Quiz: IN vs EXISTS

For a table with 10 million rows, which is typically faster?

```
A. IN (always)
B. EXISTS (always)
C. EXISTS (usually, because it short-circuits)
D. They're always identical
```

. . .

C. `EXISTS` stops at the first match. `IN` materializes the full subquery result into a list. For large datasets, that difference matters. But the optimizer is smart – sometimes it rewrites one to the other. Profile, don't guess.

4 Part 4: LATERAL Joins

The most powerful subquery pattern you've never heard of.

4.1 LATERAL Subqueries

4.2 LATERAL in FROM: Inline Calculations

`LATERAL` lets a subquery in the FROM clause reference columns from preceding tables. Think of it as a "for each row, compute this":

```sql
SELECT county_name,
       state_name,
       pop_est_2018,
       pop_est_2019,
       raw_chg,
       round(pct_chg * 100, 2) AS pct_chg
FROM us_counties_pop_est_2019,
     LATERAL (SELECT pop_est_2019 - pop_est_2018 AS raw_chg) rc,
     LATERAL (SELECT raw_chg / pop_est_2018::numeric AS pct_chg) pc
ORDER BY pct_chg DESC;
```

Run this. Each LATERAL subquery can reference columns from earlier in the FROM clause – including results from *other* LATERAL subqueries. `pct_chg` uses `raw_chg` from the first LATERAL. Without LATERAL, you'd need nested subqueries or repeat the calculation.

4.3 LATERAL with JOIN: Top-N Per Group

The killer feature: get the top N related rows for each row in the driving table.

```sql
ALTER TABLE teachers ADD CONSTRAINT id_key PRIMARY KEY (id);

CREATE TABLE teachers_lab_access (
    access_id bigint PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    access_time timestamp with time zone,
    lab_name text,
    teacher_id bigint REFERENCES teachers (id)
);

INSERT INTO teachers_lab_access (access_time, lab_name, teacher_id)
VALUES ('2022-11-30 08:59:00-05', 'Science A', 2),
       ('2022-12-01 08:58:00-05', 'Chemistry B', 2),
       ('2022-12-21 09:01:00-05', 'Chemistry A', 2),
       ('2022-12-02 11:01:00-05', 'Science B', 6),
       ('2022-12-07 10:02:00-05', 'Science A', 6),
       ('2022-12-17 16:00:00-05', 'Science B', 6);
```

Run the setup, then:

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

For each teacher, this returns their 2 most recent lab accesses. Without LATERAL, you'd need window functions + CTEs. With LATERAL, it's clean and direct.

In data engineering: "for each customer, get their last 3 orders." "For each sensor, get the most recent 5 readings." LATERAL is the tool.

4.4 Try It: Top-1 Per Teacher

Modify the LATERAL query to return only each teacher's **most recent** lab access (not two). Teachers with no lab access should still appear (show NULLs).

```sql
-- Your query here
```

...

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

Change `LIMIT 2` to `LIMIT 1`. The `LEFT JOIN` ensures all teachers appear even with no matches.

5 Part 5: Common Table Expressions (CTEs)

Named temporary result sets that make complex queries readable.

5.1 CTEs with WITH

5.2 A Simple CTE

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

The CTE filters to counties with 100k+ population. The outer query counts them by state. It's the same as a subquery, but the logic flows top-to-bottom instead of inside-out.

5.3 CTEs for Joining Aggregations

Remember the derived table join from earlier? Here it is with CTEs:

```sql
WITH
    counties (st, pop_est_2018) AS (
        SELECT state_name, sum(pop_est_2018)
        FROM us_counties_pop_est_2019
        GROUP BY state_name
    ),
    establishments (st, establishment_count) AS (
        SELECT st, sum(establishments) AS establishment_count
        FROM cbp_naics_72_establishments
        GROUP BY st
    )
SELECT counties.st,
       pop_est_2018,
       establishment_count,
       round((establishments.establishment_count /
              counties.pop_est_2018::numeric(10,1)) * 1000, 1)
           AS estabs_per_thousand
FROM counties JOIN establishments
ON counties.st = establishments.st
ORDER BY estabs_per_thousand DESC;
```

Compare this to the derived table version. Same result, dramatically more readable. Each CTE has a name and a clear purpose. In code review, this is the version that gets approved.

5.4 CTEs to Eliminate Redundancy

Remember the repeated subquery for median comparison? CTEs fix that:

```sql
WITH us_median AS (
    SELECT percentile_cont(.5)
        WITHIN GROUP (ORDER BY pop_est_2019) AS us_median_pop
    FROM us_counties_pop_est_2019
)
SELECT county_name,
       state_name AS st,
       pop_est_2019,
       us_median_pop,
       pop_est_2019 - us_median_pop AS diff_from_median
```

```sql
FROM us_counties_pop_est_2019 CROSS JOIN us_median
WHERE (pop_est_2019 - us_median_pop) BETWEEN -1000 AND 1000;
```

The median is computed once in the CTE, then CROSS JOINed to every row. No repeated subqueries. Clean, efficient, maintainable.

5.5 Try It: State-Level Summary CTE

Using two CTEs, compute:

1. CTE 1: the total population per state
2. CTE 2: the number of counties per state

Join them and add a column for average population per county. Show state, total pop, county count, and avg pop per county. Order by average descending. Which state has the largest average county?

```sql
-- Your query here
```

. . .

```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 sp
JOIN state_counties sc ON sp.state_name = sc.state_name
ORDER BY avg_pop_per_county DESC;
```

6 Part 6: Cross Tabulations

Pivot tables in SQL. Every analyst's favorite party trick.

6.1 crosstab()

6.2 Enable the Extension

```sql
CREATE EXTENSION tablefunc;
```

This loads the `crosstab()` function from the `tablefunc` module. You only need to do this once per database.

6.3 The Ice Cream Survey

```sql
SELECT *
FROM ice_cream_survey
ORDER BY response_id
LIMIT 10;
```

200 employees across offices, each picking a flavor. We want a pivot: offices as rows, flavors as columns, counts as values.

6.4 Generating the Crosstab

```sql
SELECT *
FROM crosstab(
    'SELECT office, flavor, 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 (office text,
    chocolate bigint,
    strawberry bigint,
    vanilla bigint);
```

Run this. The first argument is the source query (must have exactly 3 columns: row, category, value). The second argument defines the category labels. The `AS` clause names the output columns.

This transforms long data (one row per observation) into wide data (one row per group). Essential for reporting and dashboards.

6.5 Temperature Crosstab

A more complex example: median max temperature by station and month.

```sql
SELECT *
FROM crosstab(
    'SELECT station_name,
            date_part(''month'', observation_date),
            percentile_cont(.5)
                WITHIN GROUP (ORDER BY max_temp)
     FROM temperature_readings
     GROUP BY station_name,
              date_part(''month'', observation_date)
     ORDER BY station_name',
    'SELECT month FROM generate_series(1,12) month'
)
AS (station text,
    jan numeric(3,0), feb numeric(3,0), mar numeric(3,0),
    apr numeric(3,0), may numeric(3,0), jun numeric(3,0),
    jul numeric(3,0), aug numeric(3,0), sep numeric(3,0),
    oct numeric(3,0), nov numeric(3,0), dec numeric(3,0));
```

Run this. Two stations, 12 months, median temperatures. Notice `generate_series(1,12)` creates the month column headers. The escaped single quotes (`''`) inside the string are required.

6.6 Quick Quiz: Crosstab Requirements

The source query for `crosstab()` must return exactly how many columns?

```
A. 2
B. 3
C. 4
D. It depends on the output
```

. . .

B. Always three: (1) row identifier, (2) category, (3) value. The row identifier groups the pivot rows, the category determines which column gets the value.

7 Part 7: CASE Expressions

Conditional logic inside SQL. Your if/else for data transformation.

7.1 CASE for Classification

7.2 Reclassifying Data

```sql
SELECT max_temp,
       CASE WHEN max_temp >= 90 THEN 'Hot'
            WHEN max_temp >= 70 AND max_temp < 90 THEN 'Warm'
            WHEN max_temp >= 50 AND max_temp < 70 THEN 'Pleasant'
            WHEN max_temp >= 33 AND max_temp < 50 THEN 'Cold'
            WHEN max_temp >= 20 AND max_temp < 33 THEN 'Frigid'
            WHEN max_temp < 20 THEN 'Inhumane'
            ELSE 'No reading'
       END AS temperature_group
FROM temperature_readings
ORDER BY station_name, observation_date;
```

Each row gets classified into a group. `CASE` evaluates conditions top-to-bottom and returns the first match. `ELSE` catches anything that fell through.

7.3 CASE Inside a CTE: The Full Pattern

Combine CASE with a CTE to classify, then aggregate:

```sql
WITH temps_collapsed (station_name, max_temperature_group) AS (
    SELECT station_name,
           CASE WHEN max_temp >= 90 THEN 'Hot'
                WHEN max_temp >= 70 AND max_temp < 90 THEN 'Warm'
                WHEN max_temp >= 50 AND max_temp < 70 THEN 'Pleasant'
                WHEN max_temp >= 33 AND max_temp < 50 THEN 'Cold'
                WHEN max_temp >= 20 AND max_temp < 33 THEN 'Frigid'
                WHEN max_temp < 20 THEN 'Inhumane'
                ELSE 'No reading'
           END
    FROM temperature_readings
)
SELECT station_name, max_temperature_group, count(*)
FROM temps_collapsed
```

```sql
GROUP BY station_name, max_temperature_group
ORDER BY station_name, count(*) DESC;
```

Run this. Step 1 (CTE): classify every reading. Step 2 (outer query): count days per station per group. This two-step pattern – transform then aggregate – is the backbone of analytical queries.

7.4 Try It: Population Tiers

Using a CTE with CASE, classify each county as:

- 'Metro' (pop >= 500,000)
- 'Urban' (pop 100,000 to 499,999)
- 'Suburban' (pop 50,000 to 99,999)
- 'Rural' (pop < 50,000)

Then count the number of counties in each tier and compute the total population per tier. Which tier has the most counties? Which holds the most population?

```sql
-- Your query here
```

. . .

```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;
```

Rural has the most counties by far. Metro has the most total population. The vast majority of U.S. counties are small.

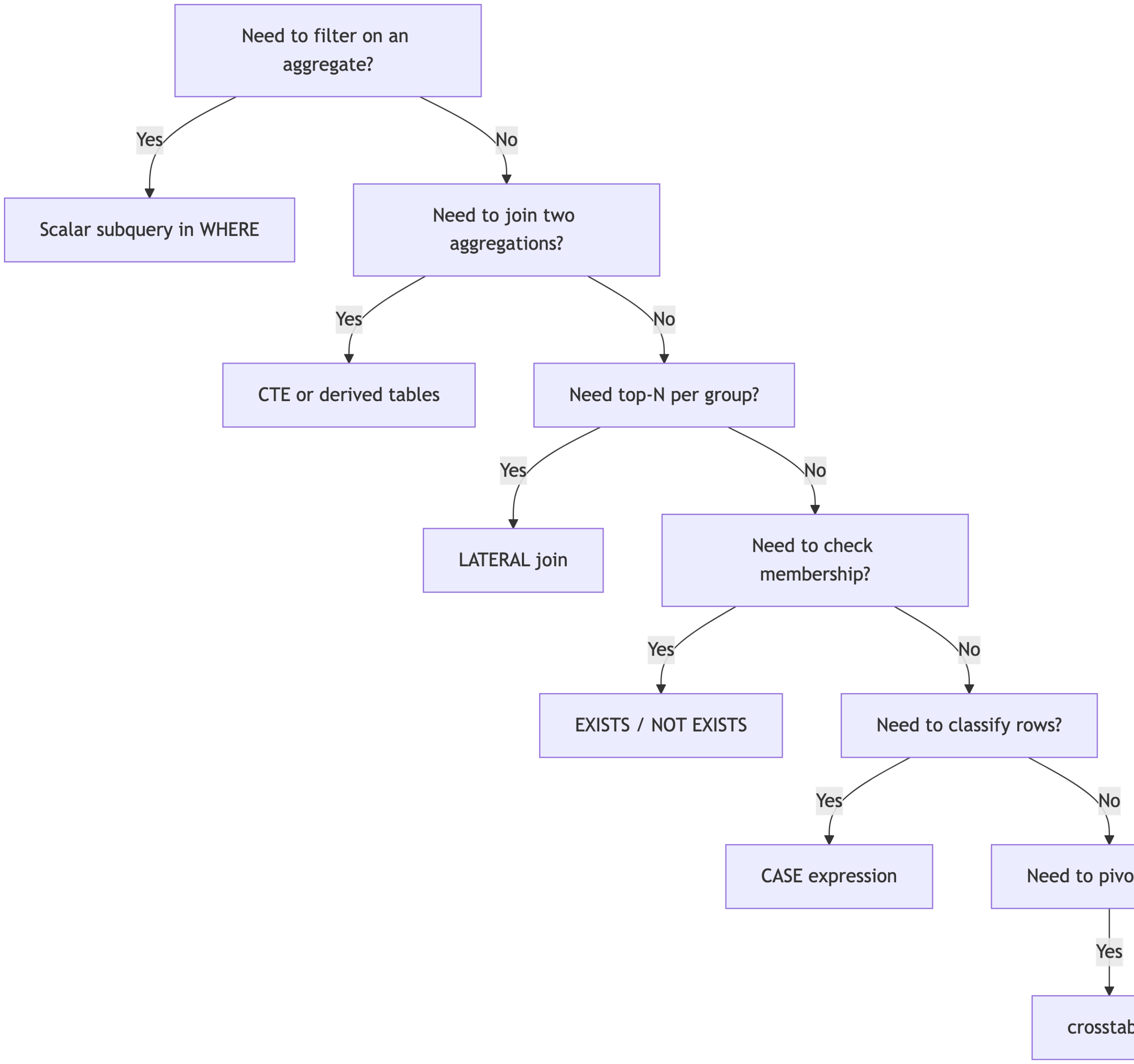
8 Part 8: Summary

8.1 What We Learned

8.2 Key Techniques

Technique	What It Does
Scalar subquery in WHERE	Compute a threshold, filter on it
Derived table (subquery in FROM)	Create a temporary table inline
Column subquery in SELECT	Add a computed column per row
Correlated subquery	Inner query references outer query (runs per row)
`IN` / `EXISTS` / `NOT EXISTS`	Check membership across tables
`LATERAL` join	Per-row subquery that can reference preceding tables
CTE (`WITH`)	Named temporary result sets for readability and reuse
`crosstab()`	Pivot long data to wide (requires `tablefunc` extension)
`CASE`	Conditional classification inside queries

8.3 When to Use What



8.4 The Big Ideas

1. **Subqueries go everywhere:** WHERE, FROM, SELECT, even inside other subqueries.
2. **CTEs > nested subqueries** for readability. Always. Fight me.
3. **LATERAL is the top-N-per-group tool.** Learn it. Love it.
4. **EXISTS short-circuits.** For large-table membership checks, prefer it over IN.
5. **CASE is your inline if/else.** Classify first, aggregate second.
6. **crosstab() pivots data.** Three columns in, pivot table out.

8.5 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 Joins
5. PostgreSQL: tablefunc (crosstab)