Advanced PostgreSQL Query Techniques

CTEs, Window Functions, Set Returning Functions & Recursive Queries

Learning Objectives

Understand Common Table Expressions (CTEs) and their applications

Master window functions for analytical queries

Utilize set-returning functions for data generation and manipulation

Implement recursive queries for hierarchical data

Apply these techniques to real-world database problems

Agenda Overview

Part 1: Common Table Expressions (CTEs)

Part 2: Window Functions

Part 3: Set Returning Functions

Part 4: Recursive Queries

Q&A and Wrap-up

Part 1

Common Table Expressions (CTEs)

What are CTEs?

Definition: Temporary named result sets that exist only during query execution

Introduced with the WITH clause

Benefits: Improved readability, modularity, and maintainability

Think of CTEs as 'query variables' or 'temporary views'

Not stored in database - evaluated at query execution time

Basic CTE Syntax

```
WITH cte_name AS (
    SELECT column1, column2
    FROM table_name
    WHERE condition
)
SELECT * FROM cte_name;

-- Example:
WITH high_earners AS (
    SELECT name, salary
    FROM employees
    WHERE salary > 100000
)
SELECT * FROM high_earners ORDER BY salary DESC;
```

CTE Example - Sales Analysis

```
WITH monthly_sales AS (
    SELECT DATE_TRUNC('month', order_date) AS month,
           SUM(amount) AS total_sales
    FROM orders
    WHERE order_date >= '2024-01-01'
    GROUP BY DATE_TRUNC('month', order_date)
)
SELECT month, total_sales,
       total_sales - LAG(total_sales)
           OVER (ORDER BY month) AS growth
FROM monthly_sales;
```

Calculates monthly sales and month-over-month growth

Multiple CTEs

```
WITH cte1 AS (
    SELECT ...
),
cte2 AS (
    SELECT ... FROM cte1 ...
),
cte3 AS (
    SELECT ... FROM cte2 ...
)
SELECT ... FROM cte3;
```

Each CTE can reference previously defined CTEs

Multiple CTEs Example

```
WITH product_sales AS (
    SELECT product_id, SUM(quantity) AS total_qty
    FROM order_items GROUP BY product_id
),
top_products AS (
    SELECT product_id, total_qty
    FROM product_sales WHERE total_qty > 100
)
SELECT p.name, tp.total_qty
FROM top_products tp
JOIN products p ON tp.product_id = p.id
ORDER BY tp.total_qty DESC;
```

CTEs vs Subqueries

Readability: CTEs are more readable for complex queries

Reusability: CTEs can be referenced multiple times

Performance: Usually similar - PostgreSQL optimizer treats them similarly

Debugging: Easier to test CTEs independently

Maintainability: CTEs better for breaking down complex logic

Readability

```sql
SELECT
    c.customer_name,
    c.total_orders,
    c.total_spent,
    a.avg_order_value
FROM (
    SELECT
        customer_id,
        customer_name,
        COUNT(*) as total_orders,
        SUM(amount) as total_spent
    FROM orders
    WHERE order_date >= '2024-01-01'
    GROUP BY customer_id, customer_name
) c
JOIN (
    SELECT AVG(order_value) as avg_order_value
    FROM (
        SELECT customer_id, SUM(amount) as order_value
        FROM orders
        WHERE order_date >= '2024-01-01'
        GROUP BY customer_id
    ) inner_orders
) a ON true
WHERE c.total_spent > a.avg_order_value;
```

```sql
WITH customer_stats AS (
    SELECT
        customer_id,
        customer_name,
        COUNT(*) as total_orders,
        SUM(amount) as total_spent
    FROM orders
    WHERE order_date >= '2024-01-01'
    GROUP BY customer_id, customer_name
),
avg_spending AS (
    SELECT AVG(total_spent) as avg_order_value
    FROM customer_stats
)
SELECT
    cs.customer_name,
    cs.total_orders,
    cs.total_spent,
    av.avg_order_value
FROM customer_stats cs
CROSS JOIN avg_spending av
WHERE cs.total_spent > av.avg_order_value;
```

Reusability

```sql
SELECT
    'High Value' as segment,
    COUNT(*) as customer_count
FROM (
    SELECT customer_id, SUM(amount) as total_spent
    FROM orders
    WHERE order_date >= '2024-01-01'
    GROUP BY customer_id
) customer_totals
WHERE total_spent > 10000

UNION ALL

SELECT
    'Medium Value' as segment,
    COUNT(*) as customer_count
FROM (
    SELECT customer_id, SUM(amount) as total_spent  -- REPEATED!
    FROM orders
    WHERE order_date >= '2024-01-01'
    GROUP BY customer_id
) customer_totals
WHERE total_spent BETWEEN 1000 AND 10000

UNION ALL

SELECT
    'Low Value' as segment,
    COUNT(*) as customer_count
FROM (
    SELECT customer_id, SUM(amount) as total_spent  -- REPEATED!
    FROM orders
    WHERE order_date >= '2024-01-01'
    GROUP BY customer_id
) customer_totals
WHERE total_spent < 1000;
```

```sql
WITH customer_totals AS (
    SELECT customer_id, SUM(amount) as total_spent
    FROM orders
    WHERE order_date >= '2024-01-01'
    GROUP BY customer_id
)
SELECT
    'High Value' as segment,
    COUNT(*) as customer_count
FROM customer_totals
WHERE total_spent > 10000

UNION ALL

SELECT 'Medium Value', COUNT(*)
FROM customer_totals
WHERE total_spent BETWEEN 1000 AND 10000

UNION ALL

SELECT 'Low Value', COUNT(*)
FROM customer_totals
WHERE total_spent < 1000;
```

Debugging

```sql
-- To test the inner subquery, you have to extract it manually
SELECT p.product_name, p.category, s.total_sales
FROM products p
JOIN (
    SELECT product_id, SUM(quantity * price) as total_sales
    FROM order_items
    WHERE order_date >= '2024-01-01'
    GROUP BY product_id
) s ON p.product_id = s.product_id
WHERE s.total_sales > 1000
ORDER BY s.total_sales DESC;
```

```sql
-- Step 1: Test this CTE independently first
WITH sales_summary AS (
    SELECT product_id, SUM(quantity * price) as total_sales
    FROM order_items
    WHERE order_date >= '2024-01-01'
    GROUP BY product_id
)
-- You can run: SELECT * FROM sales_summary; to verify

-- Step 2: Then add the full query
SELECT p.product_name, p.category, s.total_sales
FROM products p
JOIN sales_summary s ON p.product_id = s.product_id
WHERE s.total_sales > 1000
ORDER BY s.total_sales DESC;

-- To debug: Just add LIMIT or WHERE to the CTE output
```

Maintainability

```sql
SELECT
    dept_name,
    emp_count,
    avg_salary,
    high_earner_count
FROM (
    SELECT
        d.department_name as dept_name,
        COUNT(e.employee_id) as emp_count,
        AVG(e.salary) as avg_salary,
        (SELECT COUNT(*)
         FROM employees e2
         WHERE e2.department_id = d.department_id
         AND e2.salary > (
             SELECT AVG(salary) * 1.5
             FROM employees e3
             WHERE e3.department_id = d.department_id
         )) as high_earner_count
    FROM departments d
    LEFT JOIN employees e ON d.department_id = e.department_id
    GROUP BY d.department_id, d.department_name
) dept_stats
WHERE emp_count > 5;
```

```sql
WITH department_stats AS (
    -- Step 1: Get basic department statistics
    SELECT
        d.department_id,
        d.department_name,
        COUNT(e.employee_id) as emp_count,
        AVG(e.salary) as avg_salary
    FROM departments d
    LEFT JOIN employees e ON d.department_id = e.department_id
    GROUP BY d.department_id, d.department_name
),
high_earner_threshold AS (
    -- Step 2: Calculate threshold for each department
    SELECT
        department_id,
        avg_salary * 1.5 as threshold
    FROM department_stats
),
high_earners AS (
    -- Step 3: Count high earners per department
    SELECT
        e.department_id,
        COUNT(*) as high_earner_count
    FROM employees e
    JOIN high_earner_threshold t ON e.department_id = t.department_id
    WHERE e.salary > t.threshold
    GROUP BY e.department_id
)
-- Step 4: Combine all results
SELECT
    ds.department_name,
    ds.emp_count,
    ds.avg_salary,
    COALESCE(he.high_earner_count, 0) as high_earner_count
FROM department_stats ds
LEFT JOIN high_earners he ON ds.department_id = he.department_id
WHERE ds.emp_count > 5;
```

When to Use CTEs

Breaking down complex queries into logical, readable steps

When the same subquery is needed multiple times

Improving code maintainability and organization

Preparing data for recursive queries (coming later!)

Separating data preparation from final query logic

Part 2

Window Functions

Introduction to Window Functions

Perform calculations across rows related to the current row

Key difference: Unlike GROUP BY, window functions don't collapse rows

Syntax: function() OVER (partition/order clause)

Three categories: Aggregate (SUM, AVG), Ranking (RANK, ROW_NUMBER), Value (LAG, LEAD)

Window Function Syntax

```
function_name([arguments]) OVER (
    [PARTITION BY partition_expression]
    [ORDER BY sort_expression]
    [frame_clause]
)

-- PARTITION BY: Divides rows into groups
-- ORDER BY: Defines order within partition
-- Frame clause: Defines window frame (optional)
```

Basic Window Function Example

```
SELECT employee_name, department, salary,
       AVG(salary) OVER (
           PARTITION BY department
       ) AS dept_avg_salary
FROM employees;
```

Each employee row shows their department's average salary - no GROUP BY!

Ranking Functions

ROW_NUMBER(): Sequential number, no ties (1, 2, 3, 4, 5...)

RANK(): Rank with gaps for ties (1, 2, 2, 4, 5...)

DENSE_RANK(): Rank without gaps (1, 2, 2, 3, 4...)

NTILE(n): Divides rows into n groups (e.g., NTILE(4) for quartiles)

Ranking Functions Example

```
SELECT employee_name, department, salary,
       ROW_NUMBER() OVER (
           PARTITION BY department
           ORDER BY salary DESC
       ) AS row_num,
       RANK() OVER (
           PARTITION BY department
           ORDER BY salary DESC
       ) AS rank,
       DENSE_RANK() OVER (
           PARTITION BY department
           ORDER BY salary DESC
       ) AS dense_rank
FROM employees;
```

LAG and LEAD Functions

```
-- LAG: Access previous row's value
-- LEAD: Access next row's value

LAG(column, offset, default) OVER (ORDER BY ...)
LEAD(column, offset, default) OVER (ORDER BY ...)

-- offset: number of rows back/forward (default: 1)
-- default: value to return if no row exists
```

Useful for calculating differences, trends, and comparisons

LAG/LEAD Example - Sales Trends

```
SELECT month, revenue,
       LAG(revenue, 1) OVER (ORDER BY month)
           AS prev_month_revenue,
       revenue - LAG(revenue, 1) OVER (ORDER BY month)
           AS month_over_month_change,
       LEAD(revenue, 1) OVER (ORDER BY month)
           AS next_month_revenue
FROM monthly_revenue ORDER BY month;
```

Perfect for time-series analysis and trend calculation

FIRST_VALUE and LAST_VALUE

```
SELECT product_name, sale_date, price,
       FIRST_VALUE(price) OVER (
           PARTITION BY product_name
           ORDER BY sale_date
       ) AS initial_price,
       LAST_VALUE(price) OVER (
           PARTITION BY product_name
           ORDER BY sale_date
           ROWS BETWEEN UNBOUNDED PRECEDING
                    AND UNBOUNDED FOLLOWING
       ) AS final_price
FROM product_sales;
```

Note: LAST_VALUE requires careful frame specification!

Window Frame Specification

```
-- ROWS: Physical rows
-- RANGE: Logical range (values)

ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING
RANGE BETWEEN INTERVAL '1 day' PRECEDING
          AND CURRENT ROW
```

Defines subset of partition for function calculation

Moving Average Example

```
SELECT sale_date, daily_sales,
       AVG(daily_sales) OVER (
           ORDER BY sale_date
           ROWS BETWEEN 6 PRECEDING
                    AND CURRENT ROW
       ) AS seven_day_moving_avg
FROM daily_sales_summary ORDER BY sale_date;
```

Calculates 7-day moving average (current + 6 previous days)

Window Functions - Practical Use Cases

Running totals and cumulative sums

Moving averages for trend analysis

Ranking and top-N queries per group

Gap and island problems

Time-series analysis (YoY, MoM growth)

Percentile calculations (median, quartiles)

Part 3

Set Returning Functions

What are Set Returning Functions?

Functions that return multiple rows (a set) rather than a single value

Can be used in FROM clause like tables

PostgreSQL provides many built-in SRFs

Common examples: generate_series, unnest, json_array_elements, jsonb_each

generate_series()

```
-- Generate numbers 1 to 10
SELECT * FROM generate_series(1, 10);

-- Generate dates for 2024
SELECT * FROM generate_series(
    '2024-01-01'::date,
    '2024-12-31'::date,
    '1 day'::interval
);

-- Generate timestamps every hour
SELECT * FROM generate_series(
    '2024-01-01 00:00'::timestamp,
    '2024-01-01 23:00'::timestamp,
    '1 hour'::interval
);
```

generate_series() - Practical Example

```
-- Fill missing dates in sales data
WITH date_series AS (
    SELECT generate_series(
        '2024-01-01'::date,
        '2024-12-31'::date,
        '1 day'::interval
    )::date AS date
)
SELECT ds.date,
       COALESCE(s.total_sales, 0) AS total_sales
FROM date_series ds
LEFT JOIN sales s ON ds.date = s.sale_date
ORDER BY ds.date;
```

Ensures complete date range even with missing data

unnest() Function

```
-- Unnest simple array
SELECT unnest(ARRAY[1, 2, 3, 4, 5]);

-- Unnest array column
SELECT id, name, unnest(tags) AS tag
FROM articles;

-- Multiple arrays
SELECT * FROM unnest(
    ARRAY['a','b','c'],
    ARRAY[1,2,3]
) AS t(letter, number);
```

Expands an array into a set of rows

JSON Set Returning Functions

```
-- Expand JSON array to rows
SELECT json_array_elements('[1,2,3]'::json);

-- Expand JSON object to key-value pairs
SELECT * FROM jsonb_each('{"a":1, "b":2}'::jsonb);

-- Extract from nested JSON
SELECT order_id,
       elem->>'product_id' AS product_id,
       elem->>'quantity' AS quantity
FROM orders,
     jsonb_array_elements(items) AS elem;
```

Set Returning Functions - Use Cases

Generating test data and sequences

Filling gaps in time-series data

Creating calendar tables or dimension tables

Expanding arrays and JSON structures

Cross-joining for all combinations

Generating reports with complete date ranges

Part 4

Recursive Queries

Introduction to Recursive CTEs

Special type of CTE that references itself

Essential for hierarchical and tree-structured data

Two parts: Base case (anchor) and Recursive case (iteration)

UNION or UNION ALL connects the two parts

Must have termination condition to avoid infinite loops

Recursive CTE Syntax

```
WITH RECURSIVE cte_name AS (
    -- Base case (anchor member)
    SELECT ... FROM table
    WHERE initial_condition

    UNION [ALL]

    -- Recursive case (recursive member)
    SELECT ... FROM table
    JOIN cte_name ON join_condition
    WHERE termination_condition
)
SELECT * FROM cte_name;
```

Simple Recursive Example - Numbers

```
WITH RECURSIVE numbers AS (
    -- Base case: start with 1
    SELECT 1 AS n

    UNION ALL

    -- Recursive case: add 1 each time
    SELECT n + 1
    FROM numbers
    WHERE n < 10  -- Termination condition
)
SELECT * FROM numbers;
```

Output: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Organizational Hierarchy Example

```
-- Table: employees (id, name, manager_id)

WITH RECURSIVE employee_hierarchy AS (
    -- Base: CEO (no manager)
    SELECT id, name, manager_id, 1 AS level,
           name::text AS path
    FROM employees WHERE manager_id IS NULL

    UNION ALL

    -- Recursive: employees with managers
    SELECT e.id, e.name, e.manager_id,
           eh.level + 1,
           eh.path || ' -> ' || e.name
    FROM employees e
    JOIN employee_hierarchy eh ON e.manager_id = eh.id
)
SELECT * FROM employee_hierarchy ORDER BY level, name;
```

Recursive Queries - Important Considerations

Always include termination condition in WHERE clause

Be careful of infinite loops - test with small datasets first

Use UNION (not UNION ALL) to eliminate duplicates when detecting cycles

Track visited nodes with arrays to prevent cycles

PostgreSQL has max_stack_depth setting to prevent runaway queries

Combining

Techniques

Combining CTEs and Window Functions

```
WITH sales_data AS (
    SELECT product_id,
           DATE_TRUNC('month', sale_date) AS month,
           SUM(amount) AS monthly_sales
    FROM sales
    GROUP BY product_id, DATE_TRUNC('month', sale_date)
),
ranked_sales AS (
    SELECT product_id, month, monthly_sales,
           RANK() OVER (PARTITION BY month
                        ORDER BY monthly_sales DESC) AS rank,
           AVG(monthly_sales) OVER (
               PARTITION BY product_id) AS avg_product_sales
    FROM sales_data
)
SELECT * FROM ranked_sales
WHERE rank <= 5 ORDER BY month, rank;
```

Best Practices

& Tips

Performance Considerations

CTEs: PostgreSQL 12+ can inline CTEs; use MATERIALIZED/NOT MATERIALIZED hints

Window Functions: Index PARTITION BY and ORDER BY columns

Window Functions: Share computation with same WINDOW clause

Recursive Queries: Monitor recursion depth and add cycle detection

Test on production-size datasets before deploying

Debugging Tips

Test CTEs independently by querying them directly

Use EXPLAIN ANALYZE to understand query execution plans

Break complex queries into smaller, manageable CTEs

Use descriptive CTE names that describe their purpose

Comment your recursive termination logic

Test with small datasets first to verify logic

Common Pitfalls

- ❌ Forgetting termination condition in recursive queries → infinite loop
- ❌ Incorrect window frame: Not specifying UNBOUNDED for LAST_VALUE
- ❌ Not handling NULL values in window functions
- ❌ Overusing CTEs when simple subqueries would suffice
- ❌ Creating cycles: Not tracking visited nodes in graph traversal
- ❌ Ignoring performance: Not testing on production-size data

Summary

& Key Takeaways

Summary - CTEs

Temporary named result sets with WITH clause

Improve query readability and maintainability

Can be recursive for hierarchical data

Multiple CTEs can build on each other

Essential foundation for complex queries

Can be referenced multiple times in query

Summary - Window Functions

Perform calculations across row sets without grouping

Three categories: aggregate, ranking, value functions

PARTITION BY divides data, ORDER BY defines sequence

Frame specification controls calculation scope

Essential for analytics and reporting

Preserves all rows (unlike GROUP BY)

Summary - Set Returning Functions

Return multiple rows from a single function call

generate_series for sequences and date ranges

unnest for expanding arrays

JSON functions for expanding JSON structures

Powerful for data generation and transformation

Can be used in FROM clause like tables

Summary - Recursive Queries

CTEs that reference themselves

Two parts: base case (anchor) + recursive case

Essential for hierarchical and graph data

Always include termination condition

Track paths to avoid infinite loops

Use UNION for cycle detection

Practice Exercises

1. Write a CTE to find top 10 customers by total purchases in each region
2. Use window functions to calculate running totals and percentage of total
3. Create a date dimension table for 2024 using generate_series
4. Build an org chart query with recursive CTE showing employee levels
5. Combine all techniques: Cohort analysis with retention metrics by month

Additional Resources

PostgreSQL Official Documentation: www.postgresql.org/docs

Window Functions Tutorial: postgresql.org/docs/current/tutorial-window.html

Modern SQL: modern-sql.com

PostgreSQL Wiki: wiki.postgresql.org

Practice: SQLZoo, LeetCode, HackerRank SQL challenges, pgexercises.com

Questions & Discussion

Common Table Expressions (CTEs)

Window Functions

Set Returning Functions

Recursive Queries

Real-world applications and best practices

What questions do you have?

Thank You!

Contact Information & Next Steps

Practice with sample datasets

Experiment in PostgreSQL sandbox

Apply to your own projects