Advanced Aggregate Functions & Advanced Grouping Operations

Data Transformation and Aggregation: Exercises

May 18, 2025

Contents

Global Dataset for PostgreSQL

The following SQL code creates and populates the necessary tables for the exercises. Execute this script in your PostgreSQL environment before attempting the exercises.

```sql
-- Drop tables if they exist to ensure a clean setup
DROP TABLE IF EXISTS EmployeeProjects CASCADE;
DROP TABLE IF EXISTS Sales CASCADE;
DROP TABLE IF EXISTS Employees CASCADE;
DROP TABLE IF EXISTS Departments CASCADE;
DROP TABLE IF EXISTS Projects CASCADE;
DROP TABLE IF EXISTS Products CASCADE;
DROP TABLE IF EXISTS Regions CASCADE;

-- Table: Departments
CREATE TABLE Departments (
    departmentId INT PRIMARY KEY,
    departmentName VARCHAR(100) NOT NULL,
    locationCity VARCHAR(50)
);

-- Table: Employees
CREATE TABLE Employees (
    employeeId INT PRIMARY KEY,
    firstName VARCHAR(50) NOT NULL,
    lastName VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE,
    hireDate DATE NOT NULL,
    salary DECIMAL(10, 2) NOT NULL,
    departmentId INT,
    managerId INT,
    performanceScore NUMERIC(3,2), -- Score from 0.00 to 5.00
    skills TEXT[], -- Array of skills
    FOREIGN KEY (departmentId) REFERENCES Departments(departmentId),
    FOREIGN KEY (managerId) REFERENCES Employees(employeeId)
);

-- Table: Projects
CREATE TABLE Projects (
    projectId INT PRIMARY KEY,
    projectName VARCHAR(100) NOT NULL,
    startDate DATE,
    deadlineDate DATE,
    budget DECIMAL(12,2)
);

-- Table: EmployeeProjects
CREATE TABLE EmployeeProjects (
    assignmentId SERIAL PRIMARY KEY,
    employeeId INT,
    projectId INT,
    hoursWorked INT,
    taskNotes TEXT,
    FOREIGN KEY (employeeId) REFERENCES Employees(employeeId),
    FOREIGN KEY (projectId) REFERENCES Projects(projectId)
);

-- Table: Regions
CREATE TABLE Regions (
    regionId INT PRIMARY KEY,
    regionName VARCHAR(50) NOT NULL UNIQUE
);

-- Table: Products
CREATE TABLE Products (
    productId INT PRIMARY KEY,
    productName VARCHAR(100) NOT NULL,
    category VARCHAR(50),
    standardCost DECIMAL(10, 2),
    listPrice DECIMAL(10, 2)
);

```

```sql
-- Table: Sales
CREATE TABLE Sales (
    saleId INT PRIMARY KEY,
    productId INT,
    employeeId INT,
    saleDate DATE NOT NULL,
    quantity INT NOT NULL,
    regionId INT,
    notes JSONB, -- e.g., {"customerSatisfaction": 5, "followUpRequired": true}
    FOREIGN KEY (productId) REFERENCES Products(productId),
    FOREIGN KEY (employeeId) REFERENCES Employees(employeeId),
    FOREIGN KEY (regionId) REFERENCES Regions(regionId)
);

-- Insert data into Departments
INSERT INTO Departments (departmentId, departmentName, locationCity) VALUES
(1, 'Human Resources', 'New York'),
(2, 'Engineering', 'San Francisco'),
(3, 'Sales', 'Chicago'),
(4, 'Marketing', 'New York'),
(5, 'Research', 'San Francisco');

-- Insert data into Employees
INSERT INTO Employees (employeeId, firstName, lastName, email, hireDate, salary, departmentId, managerId, performanceScore, skills) VALUES
(101, 'Alice', 'Smith', 'alice.smith@example.com', '2020-01-15', 70000, 2, NULL, 4.50, ARRAY['Java', 'Python', 'SQL']),
(102, 'Bob', 'Johnson', 'bob.johnson@example.com', '2019-03-01', 80000, 2, 101, 4.20, ARRAY['Python', 'Machine Learning']),
(103, 'Carol', 'Williams', 'carol.williams@example.com', '2021-07-30', 60000, 1, NULL, 3.90, ARRAY['HR Policies', 'Recruitment']),
(104, 'David', 'Brown', 'david.brown@example.com', '2018-06-11', 95000, 2, 101, 4.80, ARRAY['Java', 'Spring Boot', 'Microservices']),
(105, 'Eve', 'Davis', 'eve.davis@example.com', '2022-01-10', 75000, 3, NULL, 4.10, ARRAY['Salesforce', 'Negotiation']),
(106, 'Frank', 'Miller', 'frank.miller@example.com', '2019-11-05', 120000, 3, 105, 4.60, ARRAY['Key Account Management', 'CRM']),
(107, 'Grace', 'Wilson', 'grace.wilson@example.com', '2020-08-20', 65000, 4, NULL, 3.70, ARRAY['SEO', 'Content Creation']),
(108, 'Henry', 'Moore', 'henry.moore@example.com', '2023-02-18', 55000, 1, 103, 4.00, ARRAY['Onboarding', 'Employee Relations']),
(109, 'Ivy', 'Taylor', 'ivy.taylor@example.com', '2017-05-25', 110000, 5, NULL, 4.90, ARRAY['Research Methodologies', 'Statistical Analysis', 'Python']),
(110, 'Jack', 'Anderson', 'jack.anderson@example.com', '2021-10-01', 72000, 5, 109, 4.30, ARRAY['Lab Techniques', 'Data Analysis']),
(111, 'Kevin', 'Spacey', 'kevin.spacey@example.com', '2020-05-15', 65000, 4, 107, 4.1, ARRAY['Digital Marketing', 'Analytics']),
(112, 'Laura', 'Palmer', 'laura.palmer@example.com', '2021-08-01', 90000, 5, 109, 4.7, ARRAY['Quantum Physics', 'Research']),
(113, 'Dale', 'Cooper', 'dale.cooper@example.com', '2019-09-10', 130000, 3, 105, 4.8, ARRAY['Strategic Sales', 'Leadership']),
(114, 'Audrey', 'Horne', 'audrey.horne@example.com', '2022-03-20', 60000, 1, 103, NULL, ARRAY['Payroll', 'Conflict Resolution']);


-- Insert data into Projects
INSERT INTO Projects (projectId, projectName, startDate, deadlineDate, budget) VALUES
(1, 'Alpha Platform', '2023-01-01', '2023-12-31', 500000),
(2, 'Beta Feature', '2023-03-15', '2023-09-30', 150000),
(3, 'Gamma Initiative', '2023-06-01', '2024-05-31', 750000),
(4, 'Delta Rollout', '2022-11-01', '2023-07-30', 300000);

-- Insert data into EmployeeProjects
INSERT INTO EmployeeProjects (employeeId, projectId, hoursWorked, taskNotes) VALUES
(101, 1, 120, 'Developed core APIs'),
(102, 1, 100, 'Machine learning model integration'),
(104, 1, 150, 'Backend services for Alpha'),
(101, 2, 80, 'API refinement for Beta feature'),
(105, 3, 200, 'Sales strategy for Gamma'),
(106, 3, 180, 'Client acquisition for Gamma'),
(107, 4, 90, 'Marketing campaign for Delta'),
(109, 2, 110, 'Research for Beta feature improvements'),
(110, 2, 70, 'Data analysis for Beta feature testing'),
(102, 3, 50, ’Consulting on ML aspects for Gamma’);

-- Insert data into Regions
INSERT INTO Regions (regionId, regionName) VALUES
(1, ’North’), (2, ’South’), (3, ’East’), (4, ’West’), (5, ’Central’);

-- Insert data into Products
INSERT INTO Products (productId, productName, category, standardCost, listPrice) VALUES
(1, ’Laptop Pro’, ’Electronics’, 800, 1200),
(2, ’Smartphone X’, ’Electronics’, 400, 700),
(3, ’Office Chair’, ’Furniture’, 100, 250),
(4, ’Desk Lamp’, ’Furniture’, 20, 45),
(5, ’Software Suite’, ’Software’, 50, 150),
(6, ’Advanced CPU’, ’Components’, 250, 400),
(7, ’Graphics Card’, ’Components’, 300, 550);

-- Insert data into Sales
INSERT INTO Sales (saleId, productId, employeeId, saleDate, quantity, regionId, notes) VALUES
(1, 1, 105, ’2022-01-20’, 2, 1, ’{"customerSatisfaction": 5, "followUpRequired": false}’),
(2, 2, 106, ’2022-02-10’, 5, 2, ’{"customerSatisfaction": 4, "discountApplied": "10%"}’),
(3, 1, 105, ’2022-02-15’, 1, 1, ’{"customerSatisfaction": 4, "followUpRequired": true, "feedback": "Needs faster shipping options"}’),
(4, 3, 106, ’2022-03-05’, 10, 3, NULL),
(5, 4, 105, ’2023-03-22’, 20, 4, ’{"customerSatisfaction": 3}’),
(6, 5, 106, ’2023-04-10’, 50, 1, ’{"customerSatisfaction": 5, "bulkOrder": true}’),
(7, 2, 105, ’2023-04-18’, 3, 2, ’{"customerSatisfaction": 5}’),
(8, 1, 106, ’2022-05-01’, 2, 3, ’{"notes": "Repeat customer"}’),
(9, 3, 105, ’2022-05-25’, 8, 4, NULL),
(10, 5, 106, ’2023-06-11’, 30, 5, ’{"customerSatisfaction": 4, "followUpRequired": true}’),
(11, 6, 102, ’2023-07-01’, 5, 1, ’{"source": "Tech Expo"}’),
(12, 7, 104, ’2023-07-05’, 3, 2, ’{"source": "Internal Purchase"}’),
(13, 1, 105, ’2022-01-25’, 3, 1, ’{"customerSatisfaction": 5}’),
(14, 2, 105, ’2023-02-12’, 2, 2, ’{"customerSatisfaction": 3, "feedback": "Item was backordered"}’),
(15, 1, 106, ’2023-01-30’, 1, 1, NULL),
(16, 3, 113, ’2022-08-15’, 12, 2, ’{"customerSatisfaction": 5}’),
(17, 4, 105, ’2022-09-01’, 25, 3, ’{"customerSatisfaction": 4, "notes": "Urgent delivery"}’),
(18, 5, 106, ’2023-08-20’, 60, 4, ’{"bulkOrder": true}’),
(19, 6, 113, ’2023-09-05’, 8, 5, NULL),
(20, 7, 105, ’2023-10-10’, 4, 1, ’{"customerSatisfaction": 5, "followUpRequired": true}’);

-- Update data for NULL examples
UPDATE Employees SET departmentId = NULL WHERE employeeId = 108; -- Henry Moore has no department
UPDATE Sales SET regionId = NULL WHERE saleId = 4; -- Sale 4 has no region
UPDATE Products SET category = NULL WHERE productId = 4; -- Desk Lamp has no category
```

Listing 1: Global Dataset for Exercises

1 Category: Advanced Aggregate Functions

1.1 STRING_AGG(expression, separator [ORDER BY ...])

1.1.1 Practice Meaning, Values, Relations, Advantages

- Problem: For each department, list the department name and a comma-separated string of its employees' first names, ordered alphabetically by first name. Employees with no department should be handled gracefully.

1.1.2 Practice Disadvantages

- Problem: What is a potential disadvantage of using STRING_AGG if the concatenated string becomes very long or if individual components need to be queried later in SQL? Show an alternative query structure if the goal is to list department names and individual employee first names for further relational processing, rather than a concatenated string.

1.1.3 Practice Inefficient Alternatives Avoidance

- Problem: A user needs to create a semicolon-separated list of all unique skills possessed by employees in the 'Engineering' department. They might consider fetching all skills and programmatically concatenating them. Show the efficient STRING_AGG approach, possibly using UNNEST if skills are in an array.

1.2 ARRAY_AGG(expression [ORDER BY ...])

1.2.1 Practice Meaning, Values, Relations, Advantages

- Problem: For each project, list the project name and an array of employeeIds of those who worked on it. The employeeIds in the array should be sorted in ascending order.

1.2.2 Practice Disadvantages

- Problem: If you use ARRAY_AGG to store a list of employee IDs for each project, what is a disadvantage if you frequently need to find projects where a specific employee ID is, for example, the *first* person assigned (first in the aggregated array)? How does this compare to a normalized structure?

1.2.3 Practice Inefficient Alternatives Avoidance

- Problem: An application needs to display each product category along with a list of all product names within that category. A naive approach might be to query all categories, then for each category, execute another query to get its products, then assemble these lists in the application. Show how ARRAY_AGG can do this efficiently in one SQL query.

1.3 JSON_AGG(expression [ORDER BY ...])

1.3.1 Practice Meaning, Values, Relations, Advantages

- Problem: For each department located in 'San Francisco', create a JSON array. Each element of the array should be a JSON object representing an employee, containing their `firstName`, `lastName`, and `salary`. Employees should be ordered by salary in descending order within the JSON array.

1.3.2 Practice Disadvantages

- Problem: What is a potential performance issue when using `JSON_AGG` to aggregate a very large number of complex objects into a single JSON array for many groups? Also, comment on type checking when consuming this JSON.

1.3.3 Practice Inefficient Alternatives Avoidance

- Problem: To create a JSON feed of products and their sales, a developer might query all products. Then, in a loop, query sales for each product and manually construct JSON strings or objects in application code. Show how `JSON_AGG` (possibly with `JSON_BUILD_OBJECT`) can produce this more directly.

1.4 PERCENTILE_CONT(fraction) WITHIN GROUP (ORDER BY sort_expression)

1.4.1 Practice Meaning, Values, Relations, Advantages

- Problem: For each product category, calculate the 25th, 50th (median), and 75th percentile of `listPrice`. Ignore products without a category.

1.4.2 Practice Disadvantages

- Problem: If `PERCENTILE_CONT` is used on a column with very few distinct values within a group (e.g., performance scores that are all integers 1, 2, 3, 4, 5), how does interpolation affect the result, and why might `PERCENTILE_DISC` sometimes be preferred in such cases?

1.4.3 Practice Inefficient Alternatives Avoidance

- Problem: To find the median salary for each department, an analyst exports all employee salaries by department to a spreadsheet, then sorts and manually finds or uses a spreadsheet function for the median for each department. Show how `PERCENTILE_CONT` simplifies this.

1.5 CORR(Y, X)

1.5.1 Practice Meaning, Values, Relations, Advantages

- Problem: Calculate the correlation coefficient between the `quantity` of products sold and their `listPrice` from the `Sales` and `Products` tables. Do this overall, not per group.

1.5.2 Practice Disadvantages

- Problem: `CORR(Y,X)` indicates the strength and direction of a linear relationship. What does a correlation coefficient near 0 imply, and what kind of strong relationship might it fail to capture?

1.5.3 Practice Inefficient Alternatives Avoidance

- Problem: To determine if there's a relationship between employee `salary` and `performanceScore`, a user exports this data for all employees into a statistical software package just to compute the Pearson correlation coefficient. Show the direct SQL method.

1.6 `REGR_SLOPE(Y, X)`

1.6.1 Practice Meaning, Values, Relations, Advantages

- Problem: For 'Electronics' products, estimate how much the average `quantity` sold changes for each one-dollar increase in `listPrice`. Use `REGR_SLOPE` considering `quantity` as Y (dependent) and `listPrice` as X (independent).

1.6.2 Practice Disadvantages

- Problem: `REGR_SLOPE(Y,X)` gives the slope of a best-fit linear line. What important information about the relationship does it *not* provide, which would be crucial for judging the reliability of this slope? (Hint: think about goodness of fit).

1.6.3 Practice Inefficient Alternatives Avoidance

- Problem: A manager wants to quickly see if higher employee salaries in the 'Sales' department are generally associated with higher performance scores by looking at the trend. They export salary and performance scores to Excel to plot them and add a linear trendline to see its slope. Show how `REGR_SLOPE` can provide this slope directly.

2 Category: Advanced Grouping Operations

2.1 `GROUPING SETS ((set1), (set2), ...)`

2.1.1 Practice Meaning, Values, Relations, Advantages

- Problem: Calculate the total sales `quantity` and sum of `listPrice` (as `totalListPriceValue`, sum of `p.listPrice * s.quantity`) with the following groupings in a single query:

  1. By `(productCategory, regionName)`
  2. By `(productCategory)` only
  3. By `(regionName)` only
  4. Grand total `()`

  Use `COALESCE` to label aggregated dimensions appropriately (e.g., 'All Categories').

2.1.2 Practice Disadvantages

- Problem: If you define many complex grouping sets, e.g., `GROUPING SETS ((a,b,c), (a,d,e), (b,f), (c,g,h,i), ...)`, what are the disadvantages in terms of query complexity and potential for user error in defining the sets?

2.1.3 Practice Inefficient Alternatives Avoidance

- Problem: A user needs total sales quantity by `(EXTRACT(YEAR FROM saleDate), category)` and also by `(EXTRACT(YEAR FROM saleDate))` only. They write two separate queries with `GROUP BY` and `UNION ALL` them. Show how `GROUPING SETS` provides a more efficient and concise solution.

2.2 `ROLLUP (col1, col2, ...)`

2.2.1 Practice Meaning, Values, Relations, Advantages

- Problem: Generate a hierarchical summary of total `hoursWorked` on projects. The hierarchy is: `departmentName` → `projectName`. Include subtotals for each department and a grand total.

2.2.2 Practice Disadvantages

- Problem: `ROLLUP(country, state, city)` generates subtotals for `(country, state, city)`, `(country, state)`, `(country)`, and `()`. What if you also need a subtotal for `(country, city)` irrespective of state, or just `(city)` total? Can `ROLLUP` do this directly, and what's the implication?

2.2.3 Practice Inefficient Alternatives Avoidance

- Problem: A manager needs a sales report showing total quantity sold, with subtotals for each `regionName`, then further subtotals for each `productCategory` within that region, and finally by `productName` within category/region. This is a clear hierarchy. An analyst unfamiliar with `ROLLUP` might try to construct this with several `UNION ALL` statements. Show the `ROLLUP` simplification.

2.3 `CUBE (col1, col2, ...)`

2.3.1 Practice Meaning, Values, Relations, Advantages

- Problem: Create a cross-tabular summary of total sales quantity (`SUM(s.quantity)`) for all possible combinations of `EXTRACT(YEAR FROM saleDate)` and `productCategory`. This should include subtotals for each year across all categories, for each category across all years, and a grand total.

2.3.2 Practice Disadvantages

- Problem: If `CUBE(colA, colB, colC, colD)` is used, it generates $2^4 = 16$ different grouping sets. What is the primary disadvantage if many of these detailed cross-totals are not actually needed by the user?

2.3.3 Practice Inefficient Alternatives Avoidance

- Problem: A user wants to explore sales data by looking at total quantities broken down by `(regionName, category)`, then by `regionName` alone, then by `category` alone, and also the grand total. Without `CUBE` (or `GROUPING SETS`), they might run four separate queries. Show how `CUBE` provides all these in one go.

3 Hardcore Combined Problem

- Problem Statement: The company wants a comprehensive report on employee project involvement and sales performance for the year 2023. The report should list:

  1. `departmentName`. If an employee has no department, list as 'No Department Assigned'.
  2. `employeeFullName` (FirstName LastName).
  3. `employeeSalary`.
  4. `salaryRankInDepartment`: Rank of the employee's salary within their department (higher salary = lower rank). Employees with the same salary get the same rank, and the next rank is skipped (standard `RANK()`). If no department, rank among other 'No Department Assigned' employees.
  5. `projectsParticipated`: A comma-separated list of distinct project names the employee participated in, ordered alphabetically by project name. If no projects, display 'None'.
  6. `totalHoursOnProjects`: Total hours worked by the employee on projects. If no projects, 0.
  7. `numberOfSalesMade2023`: Count of distinct sales made by the employee in 2023. If no sales, 0.
  8. `totalRevenueGenerated2023`: Total revenue (quantity * product listPrice) generated by the employee from sales in 2023. If no sales, 0.
  9. The report must also include department-level subtotals for `totalHoursOnProjects`, `numberOfSalesMade2023`, and `totalRevenueGenerated2023`. For these subtotal rows, `employeeFullName` should be 'Subtotal for Department', `departmentName` should be the actual department name (or 'No Department Assigned' for the group of employees without a department), and other employee-specific fields (`employeeSalary`, `salaryRankInDepartment`, `projectsParticipated`) should be NULL.
  10. Finally, a grand total row for `totalHoursOnProjects`, `numberOfSalesMade2023`, and `totalRevenueGenerated2023`. For this row, `departmentName` should be 'Grand Total', and employee-specific fields should be NULL.

  Use CTEs for modularity. Employ `RANK()` for salary ranking, `STRING_AGG` for project lists, and `GROUPING SETS` for the main aggregation including subtotals and grand total. Ensure all specified previous SQL concepts are demonstrated where appropriate (joins, basic aggregates, date functions, `COALESCE`, `CASE` expressions).