Advanced Aggregate Functions & Advanced Grouping Operations

Data Transformation and Aggregation: Solutions

May 19, 2025

Contents

Global Dataset for PostgreSQL

The following SQL code creates and populates the necessary tables for the exercises. This is the same dataset provided with 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'),
```

```sql
(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 (Identical to Exercises Document)

1 Category: Advanced Aggregate Functions

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

1.1.1 Practice Meaning, Values, Relations, Advantages

```
SELECT
    COALESCE(d.departmentName, 'No Department Assigned') AS departmentName,
    STRING_AGG(e.firstName, ', ' ORDER BY e.firstName) AS employeeNames
FROM Employees e
LEFT JOIN Departments d ON e.departmentId = d.departmentId
GROUP BY d.departmentId, d.departmentName
ORDER BY d.departmentName NULLS FIRST;
```

Listing 2: Solution 1.1.1: Department employee list

1.1.2 Practice Disadvantages

```
-- Disadvantage: STRING_AGG can produce very long strings that might exceed system limits,
-- be truncated, or be unwieldy for display or parsing. It also denormalizes data into a string,
-- making SQL operations on individual elements (e.g., finding departments with an employee
-- named 'Alice' within the aggregated string) complex and inefficient.

-- Alternative for relational processing:
SELECT
    COALESCE(d.departmentName, 'No Department Assigned') AS departmentName,
    e.firstName
FROM Employees e
LEFT JOIN Departments d ON e.departmentId = d.departmentId
ORDER BY COALESCE(d.departmentName, 'No Department Assigned') NULLS FIRST, e.firstName;
```

Listing 3: Solution 1.1.2: Disadvantage of STRING_AGG and alternative

1.1.3 Practice Inefficient Alternatives Avoidance

```
-- Inefficient: Fetching all skills for Engineering employees and then using application code
-- (e.g., Python loop) to build the unique, sorted list.

-- Efficient SQL approach using STRING_AGG with UNNEST for array 'skills' column:
SELECT
    d.departmentName,
    STRING_AGG(DISTINCT skill, '; ' ORDER BY skill) AS departmentUniqueSkills
FROM Employees e
JOIN Departments d ON e.departmentId = d.departmentId,
UNNEST(e.skills) AS skill  -- Creates a row for each skill in the array
WHERE d.departmentName = 'Engineering'
```

```sql
GROUP BY d.departmentId, d.departmentName;
```

Listing 4: Solution 1.1.3: Efficient unique skills list

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

1.2.1 Practice Meaning, Values, Relations, Advantages

```sql
SELECT
    p.projectName,
    ARRAY_AGG(ep.employeeId ORDER BY ep.employeeId) AS teamMemberIds
FROM Projects p
JOIN EmployeeProjects ep ON p.projectId = ep.projectId
GROUP BY p.projectId, p.projectName
ORDER BY p.projectName;
```

Listing 5: Solution 1.2.1: Project team member IDs array

1.2.2 Practice Disadvantages

```sql
-- Disadvantage: Querying specific positions or complex conditions within arrays
-- (e.g., "employeeId X is the first element and employeeId Y is the third")
-- can be cumbersome and less performant in SQL compared to querying normalized data.
-- Standard SQL indexes are generally not effective for searching inside array elements
-- by position or value across many rows. While PostgreSQL has good array functions,
-- complex array logic can be slower.

-- Normalized structure (EmployeeProjects table) is better for such specific relational queries:
SELECT ep.projectId, ep.employeeId -- (plus ordering or window functions if "first assigned" means based on a date or sequence)
FROM EmployeeProjects ep
ORDER BY ep.projectId, ep.assignmentId; -- Assuming assignmentId implies order

-- To find if employeeId 101 is the first in an array created by ARRAY_AGG(ep.employeeId ORDER BY ep.assignmentId):
-- SELECT p.projectName
-- FROM Projects p
-- JOIN (
--     SELECT
--         projectId,
--         ARRAY_AGG(employeeId ORDER BY assignmentId) as ids
--     FROM EmployeeProjects
--     GROUP BY projectId
-- ) AS pa ON p.projectId = pa.projectId
-- WHERE pa.ids[1] = 101;
-- This is possible but often less direct than querying normalized tables with appropriate indexing.
```

Listing 6: Solution 1.2.2: Disadvantage of ARRAY_AGG for positional queries

1.2.3 Practice Inefficient Alternatives Avoidance

```sql
-- Inefficient: N+1 queries (1 query for categories, then N queries for
    products in each category).

-- Efficient SQL approach:
SELECT
    COALESCE(p.category, 'Uncategorized') AS productCategory,
    ARRAY_AGG(p.productName ORDER BY p.productName) AS
   productNamesInCategory
FROM Products p
GROUP BY p.category
ORDER BY productCategory;
```

Listing 7: Solution 1.2.3: Efficient product names per category

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

1.3.1 Practice Meaning, Values, Relations, Advantages

```sql
SELECT
    d.departmentName,
    JSON_AGG(
        JSON_BUILD_OBJECT(
            'firstName', e.firstName,
            'lastName', e.lastName,
            'salary', e.salary
        )
        ORDER BY e.salary DESC
    ) AS employeesJson
FROM Departments d
JOIN Employees e ON d.departmentId = e.departmentId
WHERE d.locationCity = 'San Francisco'
GROUP BY d.departmentId, d.departmentName
ORDER BY d.departmentName;
```

Listing 8: Solution 1.3.1: JSON array of employees in San Francisco departments

1.3.2 Practice Disadvantages

```sql
-- Disadvantage: Generating very large JSON documents (e.g., thousands
   of employees per department,
-- each with many fields) can consume significant memory and CPU on the
    database server.
-- The resulting JSON string can also be large to transmit and parse by
    the client.
-- JSON is inherently schema-less; while JSON_BUILD_OBJECT helps
   structure it,
-- the database doesn't enforce the internal schema of the JSON object
   in the same way
-- as table columns. Consumers must implement their own validation or
   be robust to variations
-- if the JSON structure changes.
```

```sql
SELECT 'Large JSON objects from JSON_AGG can strain server resources
    and network. Type safety relies on producer/consumer discipline.' AS
     DisadvantageInfo;
```

Listing 9: Solution 1.3.2: Disadvantage of JSON_AGG for large/complex objects

1.3.3 Practice Inefficient Alternatives Avoidance

```sql
-- Inefficient: Multiple database queries (N+1 problem) and manual JSON
     construction in application code.

-- Efficient SQL approach:
SELECT
    p.productName,
    JSON_AGG(
        JSON_BUILD_OBJECT(
            'saleId', s.saleId,
            'saleDate', s.saleDate,
            'quantity', s.quantity,
            'regionName', rg.regionName,
            'saleNotes', s.notes
        )
        ORDER BY s.saleDate
    ) FILTER (WHERE s.saleId IS NOT NULL) AS salesJson -- FILTER
    ensures empty JSON array [] if no sales, not [null]
FROM Products p
LEFT JOIN Sales s ON p.productId = s.productId
LEFT JOIN Regions rg ON s.regionId = rg.regionId
GROUP BY p.productId, p.productName
ORDER BY p.productName;
```

Listing 10: Solution 1.3.3: Efficient JSON feed of products and sales

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

1.4.1 Practice Meaning, Values, Relations, Advantages

```sql
SELECT
    p.category AS productCategory,
    PERCENTILE_CONT(0.25) WITHIN GROUP (ORDER BY p.listPrice) AS
    p25ListPrice,
    PERCENTILE_CONT(0.50) WITHIN GROUP (ORDER BY p.listPrice) AS
    medianListPrice,
    PERCENTILE_CONT(0.75) WITHIN GROUP (ORDER BY p.listPrice) AS
    p75ListPrice
FROM Products p
WHERE p.category IS NOT NULL
GROUP BY p.category
ORDER BY p.category;
```

Listing 11: Solution 1.4.1: Percentiles of listPrice per category

1.4.2 Practice Disadvantages

```sql
-- Disadvantage: PERCENTILE_CONT interpolates between values to find the percentile,
-- which can result in a value not actually present in the dataset.
-- For example, if values are 1 and 2, the 0.5 percentile (median) is 1.5.
-- If the data is inherently discrete (e.g., star ratings 1-5), and you need a percentile value
-- that IS one of the actual data points, PERCENTILE_DISC might be more appropriate
-- as it picks an existing value.
-- The choice depends on whether a continuous or discrete interpretation of percentiles is desired.
SELECT 'PERCENTILE_CONT interpolates, potentially yielding values not in the dataset. For discrete data, PERCENTILE_DISC might be more intuitive if an existing value is required.' AS DisadvantageInfo;
```

Listing 12: Solution 1.4.2: Disadvantage of PERCENTILE_CONT interpolation

1.4.3 Practice Inefficient Alternatives Avoidance

```sql
-- Inefficient: Data export and manual/spreadsheet calculation for a standard statistical measure.

-- Efficient SQL approach:
SELECT
    COALESCE(d.departmentName, 'No Department Assigned') AS departmentName,
    PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY e.salary) AS medianSalary
FROM Employees e
LEFT JOIN Departments d ON e.departmentId = d.departmentId
GROUP BY d.departmentId, d.departmentName
ORDER BY medianSalary DESC NULLS LAST;
```

Listing 13: Solution 1.4.3: Efficient median salary calculation

1.5 CORR(Y, X)

1.5.1 Practice Meaning, Values, Relations, Advantages

```sql
SELECT
    CORR(s.quantity, p.listPrice) AS quantityPriceCorrelation
FROM Sales s
JOIN Products p ON s.productId = p.productId;
```

Listing 14: Solution 1.5.1: Correlation between sales quantity and list price

1.5.2 Practice Disadvantages

```sql
-- Disadvantage: CORR only measures linear association. A correlation coefficient near 0
-- means there is little to no linear relationship.
-- However, it does NOT mean there is no relationship at all. There could be a strong
-- non-linear relationship (e.g., quadratic, U-shaped) for which CORR would be close to 0.
```

```sql
-- Example: If sales quantity is high for very cheap and very expensive products but low
-- for mid-priced ones (a U-shape), CORR(quantity, price) might be near 0.
SELECT 'CORR only captures linear relationships. A value near 0 does not rule out strong non-linear relationships.' AS DisadvantageInfo;
```

Listing 15: Solution 1.5.2: Disadvantage of CORR for non-linear relationships

1.5.3 Practice Inefficient Alternatives Avoidance

```sql
-- Inefficient: Exporting data to external software for a basic correlation calculation.

-- Efficient SQL approach:
SELECT
    CORR(e.performanceScore, e.salary) AS salaryPerformanceCorrelation
FROM Employees e
WHERE e.performanceScore IS NOT NULL AND e.salary IS NOT NULL; -- CORR ignores NULL pairs
```

Listing 16: Solution 1.5.3: Efficient salary-performance correlation

1.6 REGR_SLOPE(Y, X)

1.6.1 Practice Meaning, Values, Relations, Advantages

```sql
SELECT
    p.category,
    REGR_SLOPE(s.quantity, p.listPrice) AS salesQtyVsPriceSlope
FROM Sales s
JOIN Products p ON s.productId = p.productId
WHERE p.category = 'Electronics'
GROUP BY p.category;
-- This gives the slope for individual sales transactions. For slope of average quantity
-- vs price, a subquery to average quantity per product first would be needed.
```

Listing 17: Solution 1.6.1: Regression slope for Electronics sales quantity vs. price

1.6.2 Practice Disadvantages

```sql
-- Disadvantage: REGR_SLOPE only provides the slope of the linear regression line. It does not indicate:
-- 1. The strength of the linear relationship (e.g., R-squared value, provided by REGR_R2).
--    A slope might be calculated even if data points are widely scattered and the linear model is a poor fit.
-- 2. Whether a linear model is appropriate at all (data could be non-linear).
-- 3. The statistical significance of the slope (p-value).
SELECT 'REGR_SLOPE does not indicate goodness-of-fit (like R-squared) or statistical significance of the relationship.' AS DisadvantageInfo;
```

Listing 18: Solution 1.6.2: Disadvantage of REGR_SLOPE regarding fit/significance

1.6.3 Practice Inefficient Alternatives Avoidance

```sql
-- Inefficient: Exporting data to Excel for plotting and trendline
    slope calculation for a simple linear regression.

-- Efficient SQL approach:
SELECT
    d.departmentName,
    REGR_SLOPE(e.performanceScore, e.salary) AS perfScoreVsSalarySlope
FROM Employees e
JOIN Departments d ON e.departmentId = d.departmentId
WHERE d.departmentName = 'Sales' AND e.performanceScore IS NOT NULL
GROUP BY d.departmentId, d.departmentName;
```

Listing 19: Solution 1.6.3: Efficient performance vs. salary slope for Sales dept

2 Category: Advanced Grouping Operations

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

2.1.1 Practice Meaning, Values, Relations, Advantages

```sql
SELECT
    COALESCE(p.category, 'All Categories') AS productCategory,
    COALESCE(r.regionName, 'All Regions') AS region,
    SUM(s.quantity) AS totalQuantitySold,
    SUM(p.listPrice * s.quantity) AS totalListPriceValue,
    GROUPING(p.category, r.regionName) AS groupingIndicator
FROM Sales s
JOIN Products p ON s.productId = p.productId
LEFT JOIN Regions r ON s.regionId = r.regionId
GROUP BY GROUPING SETS (
    (p.category, r.regionName),
    (p.category),
    (r.regionName),
    ()
)
ORDER BY GROUPING(p.category, r.regionName), productCategory NULLS LAST
    , region NULLS LAST;
```

Listing 20: Solution 2.1.1: Sales summary with multiple GROUPING SETS

2.1.2 Practice Disadvantages

```sql
-- Disadvantage: With many columns and/or many specific grouping sets,
    the GROUPING SETS
-- clause can become very long and complex to write correctly.
-- It increases the risk of errors in defining the desired combinations
    , or omitting a
-- necessary one / including an unnecessary one.
-- The resulting output can also be large and harder to interpret if
    not carefully
-- structured and labelled using GROUPING() and COALESCE.
-- Debugging such complex queries can be challenging.
SELECT 'Highly complex GROUPING SETS definitions can be error-prone to
    write and maintain, and their output hard to manage.' AS
    DisadvantageInfo;
```

Listing 21: Solution 2.1.2: Disadvantage of complex GROUPING SETS

2.1.3 Practice Inefficient Alternatives Avoidance

```sql
-- Inefficient/Basic (conceptual):
/*
SELECT EXTRACT(YEAR FROM s.saleDate) AS saleYear, p.category, SUM(s.
    quantity) AS totalQuantity
FROM Sales s JOIN Products p ON s.productId = p.productId
GROUP BY EXTRACT(YEAR FROM s.saleDate), p.category
UNION ALL
SELECT EXTRACT(YEAR FROM s.saleDate) AS saleYear, NULL AS category, SUM
    (s.quantity) AS totalQuantity
```

```sql
FROM Sales s JOIN Products p ON s.productId = p.productId -- Join might
    not be needed if category not selected
GROUP BY EXTRACT(YEAR FROM s.saleDate);
*/

-- Efficient GROUPING SETS approach:
SELECT
    EXTRACT(YEAR FROM s.saleDate) AS saleYear,
    COALESCE(p.category, 'All Categories for Year') AS productCategory,
    SUM(s.quantity) AS totalQuantity
FROM Sales s
JOIN Products p ON s.productId = p.productId
GROUP BY GROUPING SETS (
    (EXTRACT(YEAR FROM s.saleDate), p.category),
    (EXTRACT(YEAR FROM s.saleDate))
)
ORDER BY saleYear, productCategory NULLS LAST;
```

Listing 22: Solution 2.1.3: Efficient sales quantity by year/category with GROUPING SETS

2.2 ROLLUP (col1, col2, ...)

2.2.1 Practice Meaning, Values, Relations, Advantages

```sql
SELECT
    COALESCE(d.departmentName, 'Overall Total') AS department,
    CASE WHEN GROUPING(d.departmentName) = 0 THEN COALESCE(p.
    projectName, 'Department Total') ELSE NULL END AS project,
    SUM(ep.hoursWorked) AS totalHours,
    GROUPING(d.departmentName, p.projectName) AS groupingLevel
FROM EmployeeProjects ep
JOIN Employees e ON ep.employeeId = e.employeeId
LEFT JOIN Departments d ON e.departmentId = d.departmentId
JOIN Projects p ON ep.projectId = p.projectId
GROUP BY ROLLUP (d.departmentName, p.projectName)
ORDER BY d.departmentName NULLS LAST, p.projectName NULLS LAST;
```

Listing 23: Solution 2.2.1: Hierarchical summary of hours worked with ROLLUP

2.2.2 Practice Disadvantages

```sql
-- Disadvantage: ROLLUP is designed for strict hierarchical
    summarization. The order of
-- columns is critical and it only "rolls up" from right to left.
-- ROLLUP(country, state, city) will NOT produce a subtotal for (
    country, city) by itself
-- (skipping state) or a subtotal for (city) by itself.
-- To get such non-standard hierarchical or cross-level subtotals, you
    would need to use
-- GROUPING SETS to specify those exact combinations.
-- Relying solely on ROLLUP limits you to its predefined hierarchical
    aggregation paths.
```

```sql
SELECT 'ROLLUP is inflexible for non-hierarchical subtotals (e.g., (country, city) from ROLLUP(country, state, city)). Use GROUPING SETS for such cases.' AS DisadvantageInfo;
```

Listing 24: Solution 2.2.2: Disadvantage of ROLLUP for non-hierarchical subtotals

2.2.3 Practice Inefficient Alternatives Avoidance

```sql
-- Inefficient: Multiple UNION ALL queries to build up hierarchical subtotals.

-- Efficient ROLLUP approach:
SELECT
    COALESCE(rg.regionName, 'Grand Total') AS region,
    CASE WHEN GROUPING(rg.regionName) = 0 THEN COALESCE(p.category, 'Region Total') ELSE NULL END AS category,
    CASE WHEN GROUPING(rg.regionName, p.category) = 0 THEN COALESCE(p.productName, 'Category Total') ELSE NULL END AS productName,
    SUM(s.quantity) AS totalQuantity
FROM Sales s
JOIN Products p ON s.productId = p.productId
LEFT JOIN Regions rg ON s.regionId = rg.regionId
GROUP BY ROLLUP (rg.regionName, p.category, p.productName)
ORDER BY rg.regionName NULLS LAST, p.category NULLS LAST, p.productName NULLS LAST;
```

Listing 25: Solution 2.2.3: Efficient hierarchical sales report with ROLLUP

2.3 CUBE (col1, col2, ...)

2.3.1 Practice Meaning, Values, Relations, Advantages

```sql
SELECT
    EXTRACT(YEAR FROM s.saleDate) AS saleYear,
    COALESCE(p.category, 'All Categories') AS productCategory,
    SUM(s.quantity) AS totalQuantity,
    GROUPING(EXTRACT(YEAR FROM s.saleDate), p.category) AS groupingIndicator
FROM Sales s
JOIN Products p ON s.productId = p.productId
GROUP BY CUBE (EXTRACT(YEAR FROM s.saleDate), p.category)
ORDER BY saleYear NULLS LAST, productCategory NULLS LAST;
```

Listing 26: Solution 2.3.1: Cross-tabular sales summary with CUBE

2.3.2 Practice Disadvantages

```sql
-- Disadvantage: CUBE generates all possible combinations of subtotals (2^N for N columns).
-- If N is large, this results in a very large number of rows in the output, many of
-- which may not be relevant to the user.
-- This can lead to:
-- 1. Increased query processing time and resource consumption on the database.
```

```sql
-- 2. Large result sets that are difficult for users to navigate or for
      applications to process.
-- It's often better to use GROUPING SETS to specify only the required
    combinations if not all 2^N are needed.
SELECT 'CUBE can produce excessive, unneeded subtotals for many columns
    , leading to performance issues and unwieldy results.' AS
    DisadvantageInfo;
```

Listing 27: Solution 2.3.2: Disadvantage of CUBE generating excessive subtotals

2.3.3 Practice Inefficient Alternatives Avoidance

```sql
-- Inefficient: Running multiple separate queries or a complex UNION
    ALL of queries for each desired aggregation level.

-- Efficient CUBE approach:
SELECT
    COALESCE(r.regionName, 'All Regions') AS region,
    COALESCE(p.category, 'All Categories') AS productCategory,
    SUM(s.quantity) AS totalQuantity
FROM Sales s
LEFT JOIN Regions r ON s.regionId = r.regionId
JOIN Products p ON s.productId = p.productId
GROUP BY CUBE (r.regionName, p.category)
ORDER BY r.regionName NULLS LAST, p.category NULLS LAST;
```

Listing 28: Solution 2.3.3: Efficient multi-level sales exploration with CUBE

3 Hardcore Combined Problem

```
WITH EmployeeBasicInfo AS (
    SELECT
        e.employeeId,
        e.firstName || ' ' || e.lastName AS employeeFullName,
        EXTRACT(YEAR FROM e.hireDate) AS employeeHireYear,
        e.salary,
        e.departmentId,
        e.performanceScore,
        COALESCE(STRING_AGG(skill, ', ' ORDER BY skill), 'No Skills Listed') AS skillsList
    FROM Employees e
    LEFT JOIN UNNEST(e.skills) AS skill ON TRUE -- UNNEST skills array
    GROUP BY e.employeeId, e.firstName, e.lastName, e.hireDate, e.salary, e.departmentId, e.performanceScore
),
EmployeeProjectSummary AS (
    SELECT
        ep.employeeId,
        COALESCE(SUM(ep.hoursWorked), 0) AS totalHoursOnProjects,
        COALESCE(STRING_AGG(DISTINCT p.projectName, ', ' ORDER BY p.projectName), 'None') AS projectsParticipated
    FROM EmployeeProjects ep
    JOIN Projects p ON ep.projectId = p.projectId
    GROUP BY ep.employeeId
),
SalesWithSatisfaction2023 AS (
    SELECT
        s.employeeId,
        s.quantity,
        CAST(NULLIF(s.notes ->> 'customerSatisfaction', '') AS INTEGER) AS satisfactionScore
    FROM Sales s
    WHERE EXTRACT(YEAR FROM s.saleDate) = 2023
      AND s.employeeId IS NOT NULL
      AND s.notes ->> 'customerSatisfaction' IS NOT NULL
      AND s.notes ->> 'customerSatisfaction' ~ '^[0-9]+$' -- Ensures the string is purely numeric
),
EmployeeCorrelation2023 AS (
    SELECT
        employeeId,
        CORR(satisfactionScore, quantity) AS salesQtyVsSatisfactionCorr2023
    FROM SalesWithSatisfaction2023
    GROUP BY employeeId
    HAVING COUNT(satisfactionScore) >= 2 -- CORR needs at least 2 pairs of non-null values
),
EmployeeSalesRevenue2023 AS (
    SELECT
        s.employeeId,
        COALESCE(SUM(s.quantity * pr.listPrice), 0.00) AS totalRevenueGenerated2023
    FROM Sales s
    JOIN Products pr ON s.productId = pr.productId
    WHERE EXTRACT(YEAR FROM s.saleDate) = 2023 AND s.employeeId IS NOT
```

```sql
NULL
    GROUP BY s.employeeId
),
DepartmentalAggregates AS (
    SELECT
        e.departmentId, -- This will be NULL for employees without a department
        PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY e.salary) AS medianSalaryInDepartment,
        JSON_AGG(
            JSON_BUILD_OBJECT(
                'employeeId', e.employeeId,
                'performanceScore', e.performanceScore
            )
            ORDER BY e.performanceScore DESC
        ) FILTER (WHERE e.performanceScore IS NOT NULL) AS departmentPerformanceOverviewJson
    FROM Employees e
    GROUP BY e.departmentId -- Handles NULL departmentId as a separate group
)
SELECT
    CASE
        WHEN GROUPING(COALESCE(dpt.departmentName, 'No Department Assigned'), ebi.employeeId) = 0 THEN 'Employee Detail'
        WHEN GROUPING(ebi.employeeId) = 1 AND GROUPING(COALESCE(dpt.departmentName, 'No Department Assigned')) = 0 THEN 'Department Summary'
        ELSE 'Grand Total'
    END AS reportingLevel,

    CASE
        WHEN GROUPING(COALESCE(dpt.departmentName, 'No Department Assigned')) = 1 THEN 'Overall Summary' -- For Grand Total row
        ELSE COALESCE(dpt.departmentName, 'No Department Assigned')
    END AS departmentName,

    CASE WHEN GROUPING(ebi.employeeId) = 0 THEN ebi.employeeFullName ELSE NULL END AS employeeFullName,
    CASE WHEN GROUPING(ebi.employeeId) = 0 THEN ebi.employeeHireYear ELSE NULL END AS employeeHireYear,
    CASE WHEN GROUPING(ebi.employeeId) = 0 THEN ebi.skillsList ELSE NULL END AS skillsList,
    CASE WHEN GROUPING(ebi.employeeId) = 0 THEN COALESCE(eps.projectsParticipated, 'None') ELSE NULL END AS projectsParticipated,

    SUM(COALESCE(eps.totalHoursOnProjects, 0)) AS totalHoursOnProjects,
    SUM(COALESCE(esr.totalRevenueGenerated2023, 0.00)) AS totalRevenueGenerated2023,

    CASE WHEN GROUPING(ebi.employeeId) = 0 THEN ec.salesQtyVsSatisfactionCorr2023 ELSE NULL END AS salesQtyVsSatisfactionCorr2023,

    CASE
        WHEN GROUPING(ebi.employeeId) = 1 AND GROUPING(COALESCE(dpt.departmentName, 'No Department Assigned')) = 0 THEN da.medianSalaryInDepartment
```

```sql
        ELSE NULL
    END AS medianSalaryInDepartment,
    CASE
        WHEN GROUPING(ebi.employeeId) = 1 AND GROUPING(COALESCE(dpt.departmentName, 'No Department Assigned')) = 0 THEN COALESCE(da.departmentPerformanceOverviewJson, '[]'::JSONB)
        ELSE NULL
    END AS departmentPerformanceOverviewJson

FROM EmployeeBasicInfo ebi
LEFT JOIN Departments dpt ON ebi.departmentId = dpt.departmentId
LEFT JOIN EmployeeProjectSummary eps ON ebi.employeeId = eps.employeeId
LEFT JOIN EmployeeSalesRevenue2023 esr ON ebi.employeeId = esr.employeeId
LEFT JOIN EmployeeCorrelation2023 ec ON ebi.employeeId = ec.employeeId
LEFT JOIN DepartmentalAggregates da ON ebi.departmentId IS NOT DISTINCT FROM da.departmentId -- Handles NULL departmentId for 'No Department Assigned' group

GROUP BY GROUPING SETS (
    -- Employee Detail Level (includes department-level aggregates that will be picked by CASE based on grouping)
    (COALESCE(dpt.departmentName, 'No Department Assigned'), ebi.employeeId, ebi.employeeFullName, ebi.employeeHireYear, ebi.skillsList, eps.projectsParticipated, ec.salesQtyVsSatisfactionCorr2023, da.medianSalaryInDepartment, da.departmentPerformanceOverviewJson),
    -- Department Summary Level
    (COALESCE(dpt.departmentName, 'No Department Assigned'), da.medianSalaryInDepartment, da.departmentPerformanceOverviewJson),
    -- Grand Total Level
    ()
)
ORDER BY
    CASE
        WHEN GROUPING(COALESCE(dpt.departmentName, 'No Department Assigned')) = 1 THEN 3 -- Grand Total last
        WHEN dpt.departmentName IS NULL AND COALESCE(dpt.departmentName, 'No Department Assigned') = 'No Department Assigned' THEN 1 -- 'No Department Assigned' first
        ELSE 2 -- Actual departments
    END,
    departmentName,
    reportingLevel,
    employeeFullName NULLS LAST;
```

Listing 29: Solution to Hardcore Combined Problem