Advanced Commands: Joins and Aggregators

Complementary SQL: Exercises

May 12, 2025

Contents

Dataset for PostgreSQL

The following SQL code creates and populates the necessary tables for the exercises.

```sql
-- Dataset for PostgreSQL

-- Drop tables if they exist (for easy re-running of the script)
DROP TABLE IF EXISTS advanced_joins_aggregators.sales_data CASCADE;
DROP TABLE IF EXISTS advanced_joins_aggregators.project_assignments CASCADE;
DROP TABLE IF EXISTS advanced_joins_aggregators.projects CASCADE;
DROP TABLE IF EXISTS advanced_joins_aggregators.employees CASCADE;
DROP TABLE IF EXISTS advanced_joins_aggregators.departments CASCADE;
DROP TABLE IF EXISTS advanced_joins_aggregators.locations CASCADE;
DROP TABLE IF EXISTS advanced_joins_aggregators.job_grades CASCADE;
DROP TABLE IF EXISTS advanced_joins_aggregators.product_inventory CASCADE;
DROP TABLE IF EXISTS advanced_joins_aggregators.products CASCADE;
DROP TABLE IF EXISTS advanced_joins_aggregators.categories CASCADE;
DROP TABLE IF EXISTS advanced_joins_aggregators.product_info_natural CASCADE;
DROP TABLE IF EXISTS advanced_joins_aggregators.product_sales_natural CASCADE;
DROP TABLE IF EXISTS advanced_joins_aggregators.shift_schedules CASCADE;

-- Table Creation and Data Population

-- advanced_joins_aggregators.locations Table
CREATE TABLE advanced_joins_aggregators.locations (
    location_id SERIAL PRIMARY KEY,
    address VARCHAR(255),
    city VARCHAR(100),
    country VARCHAR(50)
);

INSERT INTO advanced_joins_aggregators.locations (address, city, country) VALUES
('123 Main St', 'New York', 'USA'),
('456 Oak Ave', 'London', 'UK'),
('789 Pine Ln', 'Tokyo', 'Japan'),
('101 Maple Dr', 'Berlin', 'Germany');

-- advanced_joins_aggregators.departments Table
CREATE TABLE advanced_joins_aggregators.departments (
    department_id SERIAL PRIMARY KEY,
    department_name VARCHAR(100) NOT NULL UNIQUE,
    location_id INT,
    creation_date DATE DEFAULT CURRENT_DATE,
    department_budget NUMERIC(15,2),
    CONSTRAINT fk_location FOREIGN KEY (location_id) REFERENCES
    advanced_joins_aggregators.locations(location_id)
);

INSERT INTO advanced_joins_aggregators.departments (department_name, location_id,
    department_budget, creation_date) VALUES
('Human Resources', 1, 500000.00, '2020-01-15'),
('Engineering', 2, 2500000.00, '2019-03-10'),
('Sales', 1, 1200000.00, '2019-06-01'),
('Marketing', 2, 800000.00, '2020-05-20'),
('Research', 3, 1500000.00, '2021-02-01'),
('Support', NULL, 300000.00, '2021-07-10'); -- Department with no location

-- advanced_joins_aggregators.employees Table
CREATE TABLE advanced_joins_aggregators.employees (
    employee_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE,
    phone_number VARCHAR(20),
    hire_date DATE NOT NULL,
    job_title VARCHAR(50),
    salary NUMERIC(10, 2) CHECK (salary > 0),
    manager_id INT,
    department_id INT,
    performance_rating INT CHECK (performance_rating BETWEEN 1 AND 5) NULL, -- 1 (Low)
    to 5 (High)
    CONSTRAINT fk_manager FOREIGN KEY (manager_id) REFERENCES advanced_joins_aggregators
    .employees(employee_id),
```

```sql
    CONSTRAINT fk_department FOREIGN KEY (department_id) REFERENCES advanced_joins_aggregators.departments(department_id)
);

INSERT INTO advanced_joins_aggregators.employees (first_name, last_name, email, phone_number, hire_date, job_title, salary, manager_id, department_id, performance_rating) VALUES
('Alice', 'Smith', 'alice.smith@example.com', '555-0101', '2019-03-01', 'CEO', 150000.00, NULL, NULL, 5), -- CEO, no manager, initially no dept
('Bob', 'Johnson', 'bob.johnson@example.com', '555-0102', '2019-06-15', 'CTO', 120000.00, 1, 2, 5),
('Charlie', 'Williams', 'charlie.williams@example.com', '555-0103', '2019-07-01', 'Lead Engineer', 90000.00, 2, 2, 4),
('Diana', 'Brown', 'diana.brown@example.com', '555-0104', '2020-01-10', 'Software Engineer', 75000.00, 3, 2, 3),
('Edward', 'Jones', 'edward.jones@example.com', '555-0105', '2020-02-20', 'Software Engineer', 72000.00, 3, 2, 4),
('Fiona', 'Garcia', 'fiona.garcia@example.com', '555-0106', '2019-09-01', 'HR Manager', 85000.00, 1, 1, 5),
('George', 'Miller', 'george.miller@example.com', '555-0107', '2021-04-15', 'HR Specialist', 60000.00, 6, 1, 3),
('Hannah', 'Davis', 'hannah.davis@example.com', '555-0108', '2019-11-01', 'Sales Director', 110000.00, 1, 3, 4),
('Ian', 'Rodriguez', 'ian.rodriguez@example.com', '555-0109', '2022-01-05', 'Sales Associate', 65000.00, 8, 3, 3),
('Julia', 'Martinez', 'julia.martinez@example.com', '555-0110', '2022-03-10', 'Sales Associate', 62000.00, 8, 3, 2),
('Kevin', 'Hernandez', 'kevin.hernandez@example.com', '555-0111', '2020-07-01', 'Marketing Head', 95000.00, 1, 4, 4),
('Laura', 'Lopez', 'laura.lopez@example.com', '555-0112', '2022-05-01', 'Marketing Specialist', 58000.00, 11, 4, 3),
('Mike', 'Gonzalez', 'mike.gonzalez@example.com', '555-0113', '2021-08-01', 'Research Scientist', 88000.00, 1, 5, 5), -- Reports to CEO
('Nina', 'Wilson', 'nina.wilson@example.com', '555-0114', '2023-01-10', 'Junior Engineer', 60000.00, 3, 2, NULL), -- New hire, no rating yet
('Oscar', 'Anderson', 'oscar.anderson@example.com', '555-0115', '2020-11-01', 'Support Lead', 70000.00, 1, 6, 4);

UPDATE advanced_joins_aggregators.employees SET department_id = 1 WHERE first_name = 'Alice'; -- Assign CEO to HR for example

-- Job Grades (for CROSS JOIN)
CREATE TABLE advanced_joins_aggregators.job_grades (
    grade_level CHAR(1) PRIMARY KEY,
    description VARCHAR(50),
    min_salary NUMERIC(10,2),
    max_salary NUMERIC(10,2)
);

INSERT INTO advanced_joins_aggregators.job_grades (grade_level, description, min_salary, max_salary) VALUES
('A', 'Entry Level', 30000, 50000),
('B', 'Junior', 45000, 70000),
('C', 'Mid-Level', 65000, 90000),
('D', 'Senior', 85000, 120000),
('E', 'Executive', 110000, 200000);

-- Shift Schedules (for CROSS JOIN)
CREATE TABLE advanced_joins_aggregators.shift_schedules (
    schedule_id SERIAL PRIMARY KEY,
    shift_name VARCHAR(50) NOT NULL,
    start_time TIME,
    end_time TIME
);
INSERT INTO advanced_joins_aggregators.shift_schedules (shift_name, start_time, end_time) VALUES
('Morning Shift', '08:00:00', '16:00:00'),
('Evening Shift', '16:00:00', '00:00:00'),
('Night Shift', '00:00:00', '08:00:00');


--  advanced_joins_aggregators.projects Table
CREATE TABLE  advanced_joins_aggregators.projects (
```

```sql
    project_id SERIAL PRIMARY KEY,
    project_name VARCHAR(100) NOT NULL,
    start_date DATE,
    end_date DATE,
    budget NUMERIC(12,2),
    department_id INT, -- Renamed from department_id_assign to department_id for NATURAL
     JOIN demo
    CONSTRAINT fk_proj_dept FOREIGN KEY (department_id) REFERENCES
    advanced_joins_aggregators.departments(department_id)
);

INSERT INTO  advanced_joins_aggregators.projects (project_name, start_date, end_date,
    budget, department_id) VALUES
('Project Alpha', '2023-01-15', '2023-12-31', 500000.00, 2),
('Project Beta', '2023-03-01', '2024-06-30', 1200000.00, 2),
('Project Gamma', '2023-05-10', '2023-11-30', 300000.00, 4),
('Project Delta', '2024-01-01', NULL, 750000.00, 5),
('Project Epsilon', '2023-02-01', '2023-08-31', 250000.00, 1);


-- Project Assignments Table
CREATE TABLE advanced_joins_aggregators.project_assignments (
    assignment_id SERIAL PRIMARY KEY,
    project_id INT,
    employee_id INT,
    role_in_project VARCHAR(50),
    assigned_date DATE,
    hours_allocated INT,
    CONSTRAINT fk_pa_project FOREIGN KEY (project_id) REFERENCES
    advanced_joins_aggregators.projects(project_id),
    CONSTRAINT fk_pa_employee FOREIGN KEY (employee_id) REFERENCES
    advanced_joins_aggregators.employees(employee_id)
);

INSERT INTO advanced_joins_aggregators.project_assignments (project_id, employee_id,
    role_in_project, assigned_date, hours_allocated) VALUES
(1, 3, 'Lead Developer', '2023-01-10', 40),
(1, 4, 'Developer', '2023-01-12', 30),
(1, 5, 'Developer', '2023-01-12', 30),
(2, 2, 'Project Manager', '2023-02-25', 20),
(2, 3, 'Senior Developer', '2023-03-01', 40),
(3, 11, 'Marketing Lead', '2023-05-05', 35),
(3, 12, 'Marketing Assistant', '2023-05-08', 25),
(4, 13, 'Lead Researcher', '2023-12-20', 40),
(5, 7, 'HR Coordinator', '2023-01-30', 15);


-- advanced_joins_aggregators.categories Table
CREATE TABLE advanced_joins_aggregators.categories (
    category_id SERIAL PRIMARY KEY,
    category_name VARCHAR(50) NOT NULL UNIQUE,
    description TEXT
);

INSERT INTO advanced_joins_aggregators.categories (category_name, description) VALUES
('Electronics', 'Devices and gadgets powered by electricity.'),
('Books', 'Printed and digital books across various genres.'),
('Clothing', 'Apparel for men, women, and children.'),
('Home Goods', 'Items for household use and decoration.'),
('Software', 'Applications and programs for computers and mobile devices.');

-- advanced_joins_aggregators.products Table
CREATE TABLE advanced_joins_aggregators.products (
    product_id SERIAL PRIMARY KEY,
    product_name VARCHAR(100) NOT NULL,
    category_id INT,
    supplier_id INT, -- Assuming a suppliers table exists, but not creating for brevity
    unit_price NUMERIC(10,2) CHECK (unit_price >= 0),
    common_code VARCHAR(10), -- For NATURAL JOIN example
    status VARCHAR(20) DEFAULT 'Active', -- For NATURAL JOIN example
    CONSTRAINT fk_prod_category FOREIGN KEY (category_id) REFERENCES
    advanced_joins_aggregators.categories(category_id)
);
```

```sql

INSERT INTO advanced_joins_aggregators.products (product_name, category_id, supplier_id,
     unit_price, common_code, status) VALUES
('Laptop Pro 15"', 1, 101, 1200.00, 'LP15', 'Active'),
('Smartphone X', 1, 102, 800.00, 'SPX', 'Active'),
('The SQL Mystery', 2, 201, 25.00, 'SQLM', 'Active'),
('Data Structures Algo', 2, 201, 45.00, 'DSA', 'Discontinued'),
('Men T-Shirt', 3, 301, 15.00, 'MTS', 'Active'),
('Women Jeans', 3, 302, 50.00, 'WJN', 'Active'),
('Coffee Maker', 4, 401, 75.00, 'CMK', 'Active'),
('Office Chair', 4, 402, 150.00, 'OCH', 'Backorder'),
('Antivirus Pro', 5, 501, 49.99, 'AVP', 'Active'),
('Photo Editor Plus', 5, 501, 89.99, 'PEP', 'Active'),
('Wireless Mouse', 1, 103, 22.50, 'WMS', 'Active'),
('History of Time', 2, 202, 18.00, 'HOT', 'Active');


-- Product Info (For NATURAL JOIN - intentional common columns)
CREATE TABLE advanced_joins_aggregators.product_info_natural (
    product_id INT PRIMARY KEY, -- Common column name 1
    common_code VARCHAR(10),    -- Common column name 2
    supplier_id INT,
    description TEXT,
    launch_date DATE
);
INSERT INTO advanced_joins_aggregators.product_info_natural (product_id, common_code,
    supplier_id, description, launch_date) VALUES
(1, 'LP15', 101, 'High-performance laptop', '2022-08-15'),
(2, 'SPX', 102, 'Latest generation smartphone', '2023-01-20'),
(3, 'SQLM', 201, 'A thrilling database mystery novel', '2021-05-10'),
(9, 'AVP', 501, 'Comprehensive antivirus solution', '2022-01-01');

-- Product Sales (For NATURAL JOIN - intentional common columns)
CREATE TABLE advanced_joins_aggregators.product_sales_natural (
    sale_id SERIAL PRIMARY KEY,
    product_id INT,          -- Common column name 1
    common_code VARCHAR(10), -- Common column name 2
    sale_date DATE,
    quantity_sold INT,
    customer_id_text VARCHAR(10) -- Using different name to avoid auto-join if it
    existed elsewhere
);
INSERT INTO advanced_joins_aggregators.product_sales_natural (product_id, common_code,
    sale_date, quantity_sold, customer_id_text) VALUES
(1, 'LP15', '2023-10-01', 5, 'CUST001'),
(2, 'SPX', '2023-10-05', 10, 'CUST002'),
(1, 'LP15', '2023-10-10', 3, 'CUST003'),
(9, 'AVP', '2023-11-01', 20, 'CUST004');


-- Sales Data Table (For Aggregators)
CREATE TABLE advanced_joins_aggregators.sales_data (
    sale_id SERIAL PRIMARY KEY,
    product_id INT,
    employee_id INT, -- Salesperson
    customer_id_text VARCHAR(10), -- Simulating a customer identifier
    sale_date TIMESTAMP,
    quantity_sold INT CHECK (quantity_sold > 0),
    unit_price_at_sale NUMERIC(10,2) CHECK (unit_price_at_sale >= 0),
    discount_percentage NUMERIC(4,2) DEFAULT 0 CHECK (discount_percentage BETWEEN 0 AND
    1),
    region VARCHAR(50), -- e.g., 'North America', 'Europe', 'Asia'
    payment_method VARCHAR(20), -- e.g., 'Credit Card', 'PayPal', 'Cash'
    CONSTRAINT fk_sd_product FOREIGN KEY (product_id) REFERENCES
    advanced_joins_aggregators.products(product_id),
    CONSTRAINT fk_sd_employee FOREIGN KEY (employee_id) REFERENCES
    advanced_joins_aggregators.employees(employee_id)
);

INSERT INTO advanced_joins_aggregators.sales_data (product_id, employee_id,
    customer_id_text, sale_date, quantity_sold, unit_price_at_sale, discount_percentage,
     region, payment_method) VALUES
```

```
(1, 9, 'CUST001', '2023-01-15 10:30:00', 1, 1200.00, 0.05, 'North America', 'Credit Card
    '),
(2, 10, 'CUST002', '2023-01-20 14:00:00', 2, 800.00, 0.0, 'Europe', 'PayPal'),
(3, 9, 'CUST003', '2023-02-01 09:15:00', 5, 25.00, 0.1, 'Asia', 'Credit Card'),
(5, 10, 'CUST001', '2023-02-10 11:00:00', 3, 15.00, 0.0, 'North America', 'Cash'),
(7, 9, 'CUST004', '2023-03-05 16:45:00', 1, 75.00, 0.0, 'Europe', 'Credit Card'),
(9, 10, 'CUST002', '2023-03-12 10:00:00', 2, 49.99, 0.02, 'North America', 'PayPal'),
(10, 9, 'CUST005', '2023-04-01 13:20:00', 1, 89.99, 0.0, 'Asia', 'Credit Card'),
(1, 8, 'CUST006', '2023-04-10 09:00:00', 1, 1200.00, 0.1, 'Europe', 'Credit Card'), --
    High perf employee (Hannah)
(4, 10, 'CUST001', '2023-05-01 17:00:00', 10, 45.00, 0.15, 'North America', 'Cash'), --
    Large sale value
(6, 9, 'CUST007', '2023-05-15 11:30:00', 2, 50.00, 0.0, 'Europe', 'PayPal'),
(8, 10, 'CUST003', '2023-06-01 10:10:00', 1, 150.00, 0.05, 'Asia', 'Credit Card'),
(11, 8, 'CUST008', '2023-06-10 14:30:00', 4, 22.50, 0.0, 'North America', 'Credit Card')
    , -- High perf employee (Hannah)
(12, 9, 'CUST004', '2023-06-20 15:00:00', 3, 18.00, 0.0, 'Europe', 'Cash'),
(1, 10, 'CUST005', '2023-07-01 09:45:00', 1, 1150.00, 0.0, 'North America', 'PayPal'),
    -- Slightly lower price
(2, 8, 'CUST001', '2023-07-05 12:00:00', 1, 790.00, 0.0, 'Europe', 'Credit Card'), --
    High perf employee (Hannah), high value
(3, 9, 'CUST002', '2023-01-17 10:30:00', 1, 25.00, 0.0, 'North America', 'Credit Card'),
     -- Same customer, different product
(5, 10, 'CUST003', '2023-02-15 11:00:00', 2, 15.00, 0.0, 'Asia', 'Cash'), -- Same
    customer
(7, 9, 'CUST001', '2023-03-08 16:45:00', 3, 70.00, 0.0, 'North America', 'Credit Card');
     -- Same customer, high value sale > 200
```

Listing 1: Dataset for Joins and Aggregators Exercises

1 Joins (CROSS JOIN, NATURAL JOIN, SELF JOIN, USING clause)

1.1 (i) Practice meanings, values, relations, advantages of all its technical concepts

Exercise 1.1.1: (CROSS JOIN - Meaning & Advantage)
Problem: The company wants to create a list of all possible pairings of employee first names and available shift schedules to evaluate potential staffing options. Display the employee's first name and the shift name for every combination.

Exercise 1.1.2: (NATURAL JOIN - Meaning & Advantage)
Problem: List all projects and their corresponding department names. The `projects` table has a `department_id` column, and the `departments` table also has a `department_id` column (which is its primary key). Use the most concise join syntax available for this specific scenario where column names are identical and represent the join key.

Exercise 1.1.3: (SELF JOIN - Meaning & Advantage)
Problem: Display a list of all employees and the first and last name of their respective managers. Label the manager's name columns as `manager_first_name` and `manager_last_name`. Include employees who do not have a manager (their manager's name should appear as NULL).

Exercise 1.1.4: (USING clause - Meaning & Advantage)
Problem: List all employees (first name, last name) and the name of the department they belong to. Use the `USING` clause for the join condition, as both `employees` and `departments` tables share a `department_id` column for this relationship.

1.2 (ii) Practice entirely their disadvantages of all its technical concepts

Exercise 1.2.1: (CROSS JOIN - Disadvantage)
Problem: You were asked to get a list of employees and their department names. By mistake, you wrote a query that might produce an extremely large, unintended result if not for the small size of the sample `job_grades` table. Write this problematic query using `employees` and `job_grades` and explain the disadvantage. Then, show how many rows it would produce if `employees` had 1,000 rows and `job_grades` had 10 rows.

Exercise 1.2.2: (NATURAL JOIN - Disadvantage)
Problem: The `product_info_natural` table and `product_sales_natural` table both have `product_id` and `common_code` columns. Demonstrate how using `NATURAL JOIN` between them can lead to unexpected results or errors if the assumption about common columns is incorrect or changes. Assume you only intended to join on `product_id`. What happens if `common_code` values differ for the same `product_id` or if another common column is added later?

Exercise 1.2.3: **(SELF JOIN - Disadvantage)**
Problem: When writing a query to find employees and their managers, if not careful, a `SELF JOIN` can become complex to read or write, especially with multiple levels of hierarchy or if the aliases are not clear. Illustrate a slightly more complex (but still basic) self-join requirement: Find employees who earn more than their direct manager. Point out how the logic, while powerful, could be misconstrued if not read carefully.

Exercise 1.2.4: **(USING clause - Disadvantage)**
Problem: Suppose you want to join `employees` and `departments` but also need to apply a condition on the `department_id` from a specific table (e.g., `employees.department_id = 1`) within the `ON` clause for some complex logic (not a simple post-join `WHERE`). Show why `USING(department_id)` might be less flexible or insufficient for such a scenario compared to an `ON` clause.

1.3 (iii) Practice entirely cases where people in general does not use these approaches losing their advantages, relations and values because of the easier, basic, common or easily understandable but highly inefficient solutions

Exercise 1.3.1: **(CROSS JOIN - Inefficient Alternative)**
Problem: A junior developer needs to generate all possible pairings of 3 specific employees ('Alice Smith', 'Bob Johnson', 'Charlie Williams') with all available shift schedules. Instead of using `CROSS JOIN`, they write three separate queries and plan to combine the results manually in their application or using `UNION ALL`. Show this inefficient approach and then the efficient `CROSS JOIN` solution.

Exercise 1.3.2: **(NATURAL JOIN - Avoiding for "Safety" by being overly verbose)**
Problem: A developer needs to join `product_info_natural` and `product_sales_natural`. They know both tables have `product_id` and `common_code` and they intend to join on both. They avoid `NATURAL JOIN` due to general warnings about its use and instead write a verbose `INNER JOIN ON` clause. Show this verbose solution and then the concise `NATURAL JOIN` (acknowledging that in this *specific* case, if the intent is to join on *all* common columns, `NATURAL JOIN` is concise, though still risky for future changes).

Exercise 1.3.3: **(SELF JOIN - Inefficient Alternative: Multiple Queries)**
Problem: To get each employee's name and their manager's name, a developer decides to first fetch all employees. Then, for each employee with a `manager_id`, they run a separate query to find that manager's name. Describe this highly inefficient N+1 query approach and contrast it with the efficient `SELF JOIN`.

Exercise 1.3.4: **(USING clause - Inefficient Alternative: Always typing full ON clause)**
Problem: A developer needs to join `employees` and `departments` on `department_id`. Both tables have this column name. Instead of the concise `USING(department_id)`, they always write the full `ON e.department_id = d.department_id`. While not performance-inefficient, discuss how this makes the query longer and potentially misses a small readability/maintenance advantage of `USING`.

1.4 (iv) Practice a hardcore problem combining all the technical concepts

Exercise 1.4.1: (Joins - Hardcore Problem)

Problem: The company wants a detailed report to identify "High-Impact Managers" in departments located in the 'USA'. A "High-Impact Manager" is defined as a manager who:

a. Works in a department located in the 'USA'.

b. Was hired on or before '2020-01-01'.

c. Manages at least 2 employees.

d. The average salary of their direct reports is greater than $65,000.

The report should list:

- Manager's full name (`manager_name`).
- Manager's job title (`manager_job_title`).
- Manager's department name (`department_name`).
- The city of the department (`department_city`).
- The number of direct reports (`num_direct_reports`).
- The average salary of their direct reports (`avg_reports_salary`), formatted to 2 decimal places.

Additionally:

- Order the results by the manager's last name.
- If a manager could be listed due to managing employees in multiple departments (not applicable with current schema but consider if structure allowed it), they should be listed per department criteria.
- This problem primarily tests SELF JOINs (for manager-employee hierarchy), standard JOINs (employees to departments, departments to locations), subqueries or CTEs for aggregation, and filtering with WHERE clause (Basic SQL, Date Functions, Arithmetic). While CROSS JOIN and NATURAL JOIN are not central to the optimal solution, briefly comment on whether a NATURAL JOIN between `employees` and `departments` (if `department_id` was the only common column) or `departments` and `projects` (as `department_id` is common) would have been suitable and its risks.

2 Aggregators (COUNT(DISTINCT), FILTER clause)

2.1 (i) Practice meanings, values, relations, advantages of all its technical concepts

Exercise 2.1.1: (COUNT(DISTINCT column) - Meaning & Advantage)
Problem: The sales department wants to know how many unique customers have made purchases from the `sales_data` table.

Exercise 2.1.2: (FILTER clause - Meaning & Advantage)
Problem: Calculate the total number of sales transactions and, in the same query, the number of sales transactions specifically made in the 'Europe' region. Use the `FILTER` clause for the conditional count.

2.2 (ii) Practice entirely their disadvantages of all its technical concepts

Exercise 2.2.1: (COUNT(DISTINCT column) - Disadvantage)
Problem: Explain a potential performance disadvantage of using `COUNT(DISTINCT column)` on a very large table, especially if the column is not well-indexed or has high cardinality. Why might it be slower than `COUNT(*)`?

Exercise 2.2.2: (FILTER clause - Disadvantage)
Problem: While the `FILTER` clause is standard SQL, what could be a practical disadvantage if you are working with an older version of a specific RDBMS that doesn't support it, or if you need to write a query that is portable across RDBMS versions, some of which might not support `FILTER`? What would be the alternative in such cases?

2.3 (iii) Practice entirely cases where people in general does not use these approaches losing their advantages, relations and values because of the easier, basic, common or easily understandable but highly inefficient solutions

Exercise 2.3.1: (COUNT(DISTINCT column) - Inefficient Alternative)
Problem: A data analyst needs to find the number of unique products sold. Instead of using `COUNT(DISTINCT product_id)`, they first select all distinct product IDs into a subquery and then count the rows from that subquery. Show this less direct (and potentially less optimized by some older DBs) approach.

Exercise 2.3.2: (FILTER clause - Inefficient Alternative: Multiple Queries or Complex CASE)
Problem: An analyst needs to count sales: total sales, sales in 'North America', and sales paid by 'PayPal'. Instead of using `FILTER`, they write three separate queries or use multiple `SUM(CASE WHEN ... THEN 1 ELSE 0 END)` expressions which can be less readable for simple counts. Show the multiple query approach (conceptually) and the `SUM(CASE...)` approach, then the `FILTER` clause solution.

2.4 (iv) Practice a hardcore problem combining all the technical concepts

Exercise 2.4.1: (Aggregators - Hardcore Problem)

Problem: Generate a sales performance report for product categories. The report should include, for each product category:

a. `category_name`: The name of the product category.

b. `total_revenue`: Total revenue generated for the category. Revenue for a sale item is `(quantity_sold * unit_price_at_sale * (1 - discount_percentage))`. Format to 2 decimal places.

c. `unique_customers_count`: The number of unique customers who purchased products in this category. (Uses `COUNT(DISTINCT)`).

d. `high_perf_employee_sales_count`: The number of sales transactions in this category handled by 'High-Performance' employees (defined as employees with `performance_rating = 5`). (Uses `FILTER`).

e. `high_value_cc_sales_usa_count`: The number of sales transactions in this category that had a total value (`quantity_sold * unit_price_at_sale`) over $200, were made in the 'North America' region, AND were paid by 'Credit Card'. (Uses `FILTER`).

f. `category_revenue_rank`: The rank of the category based on `total_revenue` in descending order. Use `DENSE_RANK()`.

Filtering Criteria for Output:

- Only include categories where `high_perf_employee_sales_count` is at least 1.
- AND the `unique_customers_count` is greater than 2.

Output Order:

- Order the final result by `category_revenue_rank` (ascending), then by `category_name`.

Required Concepts:

- `COUNT(DISTINCT)`
- `FILTER` clause for conditional aggregation.
- `JOINs` (products to categories, sales_data to products, sales_data to employees).
- Basic aggregators (`SUM`).
- `GROUP BY` category.
- `HAVING` clause for filtering groups based on aggregated values.
- Window Functions (`DENSE_RANK()`).
- Arithmetic operations.
- String formatting for revenue.
- Subqueries or CTEs if they simplify the logic.