Advanced Commands: Joins and Aggregators

Complementary SQL: Solutions

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 Creation 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

1.1.1 Exercise 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.

Solution:

```sql
SELECT
    e.first_name,
    ss.shift_name
FROM
    employees e
CROSS JOIN
    shift_schedules ss
ORDER BY
    e.first_name, ss.shift_name;
```

Explanation: This query demonstrates `CROSS JOIN`. Its meaning is to produce a Cartesian product of the two tables: every row from `employees` is combined with every row from `shift_schedules`. The advantage here is generating a comprehensive list of all potential assignments, which can be useful for planning or combinatorial analysis.

1.1.2 Exercise 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.

Solution:

```sql
SELECT
    p.project_name,
    d.department_name
FROM
    projects p
NATURAL JOIN
    departments d -- Joins on common column: department_id
ORDER BY
    p.project_name;
```

Explanation: `NATURAL JOIN` automatically joins tables on all columns that have the same name and compatible data types. Here, both `projects` and `departments` have `department_id`. The advantage is conciseness (no `ON` or `USING` clause needed). It's related to `INNER JOIN` as it performs an inner join based on the common columns. *Note: While concise, `NATURAL JOIN` should be used with caution (see disadvantages).*

1.1.3 Exercise 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).

Solution:

```sql
SELECT
    e.first_name AS employee_first_name,
    e.last_name AS employee_last_name,
    m.first_name AS manager_first_name,
    m.last_name AS manager_last_name
FROM
    employees e
LEFT JOIN
    employees m ON e.manager_id = m.employee_id
ORDER BY
    e.last_name, e.first_name;
```

Explanation: This is a SELF JOIN because the employees table is joined to itself. It's achieved by using table aliases (e for employee, m for manager). The meaning is to relate rows within the same table based on a hierarchical relationship (employee-manager). The advantage is the ability to query such hierarchical data or compare rows within the same table without needing separate tables for different roles (like a dedicated managers table). A LEFT JOIN is used to include employees without managers (like the CEO).

1.1.4 Exercise 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.

Solution:

```sql
SELECT
    e.first_name,
    e.last_name,
    d.department_name
FROM
    employees e
INNER JOIN
    departments d USING (department_id) -- department_id is common to both
ORDER BY
    d.department_name, e.last_name, e.first_name;
```

Explanation: The USING clause is a shorthand for an ON clause when the columns to be joined have the same name in both tables. Its meaning is to specify the join column(s) by name. The advantage is conciseness compared to ON e.department_id = d.department_id and it also implies that the join column will appear only once in the output if SELECT * were used (though behavior can vary slightly across RDBMS for SELECT *). It's clearer than NATURAL JOIN if there are multiple common columns but you only want to join on specific ones.

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

1.2.1 Exercise 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.

Solution:

```sql
-- Problematic Query (Intentional misuse for demonstration)
SELECT
    e.first_name,
    jg.grade_level,
    jg.description
FROM
    employees e
CROSS JOIN
    job_grades jg;

-- Count of rows from the query on sample data:
SELECT COUNT(*) FROM employees CROSS JOIN job_grades;
-- If employees had 1,000 rows and job_grades had 10 rows, it would produce 1000 * 10 = 10,000 rows.
```

Explanation: The primary disadvantage of `CROSS JOIN` is that it generates a Cartesian product. If used unintentionally (e.g., forgetting a `WHERE` clause in older implicit join syntax, or by mistake with explicit `CROSS JOIN`), it can lead to massive result sets that consume significant server resources and time. In this example, every employee is paired with every job grade, which is likely not the intended result for a typical "employee and their department" type query. The number of rows is `COUNT(table1) * COUNT(table2)`.

1.2.2 Exercise 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?

Solution:

```sql
-- Scenario: Joining product_info_natural and product_sales_natural
SELECT *
FROM product_info_natural
NATURAL JOIN product_sales_natural;

-- To show what it's actually joining on:
-- The NATURAL JOIN above is equivalent to:
SELECT *
FROM product_info_natural pin
INNER JOIN product_sales_natural psn
  ON pin.product_id = psn.product_id AND pin.common_code = psn.common_code;
```

Explanation: Disadvantages of `NATURAL JOIN`:

1. **Hidden Join Columns:** The join condition is implicit. If tables share multiple column names (`product_id` AND `common_code` here), `NATURAL JOIN` will use all of them. This might not be the intention (e.g., if you only wanted to join on `product_id`). If `common_code` for a given `product_id` doesn't match between the tables, that product pairing will be excluded, leading to missing data.

2. **Fragility to Schema Changes:** If a new column with the same name is added to both tables in the future (e.g., `status_code`), `NATURAL JOIN` will automatically include it in the join condition, potentially breaking the query or leading to incorrect results without any explicit change to the query itself.

3. **Readability/Maintainability:** It can be harder for someone else (or your future self) to understand the exact join logic without inspecting table schemas. Explicit `ON` or `USING` clauses are generally clearer and safer.

For example, if `product_info_natural` had `(1, 'LP15X', ...)` and `product_sales_natural` had `(1, 'LP15', ...)` for `product_id` 1, they would not join despite matching `product_id`, because `common_code` differs.

1.2.3 Exercise 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.

Solution:

```sql
SELECT
    e.first_name || ' ' || e.last_name AS employee_name,
    e.salary AS employee_salary,
    m.first_name || ' ' || m.last_name AS manager_name,
    m.salary AS manager_salary
FROM
    employees e
INNER JOIN -- Use INNER JOIN because we need manager's salary for comparison
    employees m ON e.manager_id = m.employee_id
WHERE
    e.salary > m.salary;
```

Explanation: Disadvantages of `SELF JOIN`:

1. **Readability:** For those unfamiliar with the concept, joining a table to itself can be confusing. The use of aliases is crucial, and if not chosen well, can make the query hard to follow (e.g., `t1` and `t2` are less descriptive than `e` and `m`).

2. **Complexity:** As requirements grow (e.g., finding grand-managers, or specific paths in a hierarchy), self-join queries can become quite complex and challenging to debug.

3. **Potential for Errors:** It's easy to make mistakes in the join condition (e.g., `e.employee_id = m.manager_id` instead of `e.manager_id = m.employee_id`), leading to incorrect results. In this example, the `WHERE e.salary > m.salary` clearly

states the condition, but understanding which alias refers to whom requires careful reading of the ON clause.

1.2.4 Exercise 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.

Solution:

```sql
-- Attempting with USING (and failing to add complex condition directly in join)
-- SELECT e.first_name, d.department_name
-- FROM employees e
-- INNER JOIN departments d USING (department_id)
-- -- WHERE e.department_id = 1; -- This is a post-join filter

-- Using ON for more flexibility (hypothetical complex condition)
SELECT
    e.first_name,
    d.department_name
FROM
    employees e
INNER JOIN
    departments d ON e.department_id = d.department_id AND e.salary > 60000;
    -- The "AND e.salary > 60000" is part of the join condition here.
    -- While often equivalent to a WHERE clause for INNER JOIN,
    -- for OUTER JOINs, conditions in ON vs WHERE behave differently.
    -- USING does not allow such additional non-equijoin conditions within its syntax.

-- Example of where USING is restrictive:
-- If you needed to join on department_id BUT also ensure the employee's specific department_id was, say, part of an active list from another subquery.
-- ON e.department_id = d.department_id AND e.department_id IN (SELECT active_dept_id FROM some_other_table)
-- This cannot be expressed with USING(department_id) directly.
```

Explanation: Disadvantages of `USING`:

1. **Requires Identical Column Names:** It only works if the join columns have exactly the same name in both tables. If they differ (e.g., `emp_dept_id` and `dept_id`), you must use `ON`.

2. **Less Flexible for Complex Conditions:** The `ON` clause allows for more complex join conditions, including non-equijoins (e.g., `ON e.salary BETWEEN jg.min_salary AND jg.max_salary`) or additional conditions related to the join columns themselves (as shown in the example, like `AND e.some_flag = TRUE`). `USING` is strictly for equi-joins on identically named columns.

3. **Ambiguity with `SELECT *`:** While `USING` de-duplicates the join column in `SELECT *` (usually), relying on `SELECT *` is generally bad practice. Explicitly selecting

columns is better, making this advantage of USING less critical. The main point is the restriction on join condition complexity.

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

1.3.1 Exercise 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.

Solution: Inefficient Approach (Conceptual - actual queries would be verbose):

```sql
-- Inefficient approach: Multiple queries
SELECT 'Alice' AS first_name, 'Smith' AS last_name, ss.shift_name
FROM shift_schedules ss;
-- (Developer would then run similar queries for Bob and Charlie)

-- More explicitly, using UNION ALL:
SELECT e.first_name, e.last_name, ss.shift_name
FROM employees e, shift_schedules ss
WHERE e.first_name = 'Alice' AND e.last_name = 'Smith'
UNION ALL
SELECT e.first_name, e.last_name, ss.shift_name
FROM employees e, shift_schedules ss
WHERE e.first_name = 'Bob' AND e.last_name = 'Johnson'
UNION ALL
SELECT e.first_name, e.last_name, ss.shift_name
FROM employees e, shift_schedules ss
WHERE e.first_name = 'Charlie' AND e.last_name = 'Williams';

-- Highly inefficient if done in application code by fetching all
    employees then all shifts and looping.
-- For example (pseudo-code):
-- employees_list = query("SELECT first_name, last_name FROM employees
    WHERE (first_name='Alice' AND last_name='Smith') OR ...")
-- shifts_list = query("SELECT shift_name FROM shift_schedules")
-- FOR EACH emp IN employees_list:
--   FOR EACH shift IN shifts_list:
--     PRINT emp.first_name, shift.shift_name
```

Efficient CROSS JOIN Solution:

```sql
SELECT
    e.first_name,
    e.last_name,
    ss.shift_name
FROM
    employees e
CROSS JOIN
    shift_schedules ss
```

```sql
WHERE
    (e.first_name = 'Alice' AND e.last_name = 'Smith') OR
    (e.first_name = 'Bob' AND e.last_name = 'Johnson') OR
    (e.first_name = 'Charlie' AND e.last_name = 'Williams')
ORDER BY
    e.last_name, e.first_name, ss.shift_name;
```

Explanation: The inefficient approach involves multiple database queries or fetching larger-than-needed datasets to the client for manual combination. This increases network traffic, database load, and application complexity. `CROSS JOIN` (with appropriate filtering if only a subset of one table is needed) achieves the Cartesian product directly and efficiently within the database. The `UNION ALL` method is better than client-side loops but still more verbose and potentially less optimized by the RDBMS than a direct `CROSS JOIN` on filtered inputs.

1.3.2 Exercise 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).

Solution: Verbose but "Safer" `INNER JOIN ON`:

```sql
SELECT
    pi.product_id,
    pi.common_code,
    pi.description,
    ps.sale_date,
    ps.quantity_sold
FROM
    product_info_natural pi
INNER JOIN
    product_sales_natural ps ON pi.product_id = ps.product_id AND pi.
   common_code = ps.common_code;
```

Concise `NATURAL JOIN` (if intention matches exactly):

```sql
SELECT
    product_id,        -- Note: common columns appear once
    common_code,       -- Note: common columns appear once
    description,
    sale_date,
    quantity_sold
FROM
    product_info_natural
NATURAL JOIN
    product_sales_natural;
```

Explanation: Some developers avoid `NATURAL JOIN` entirely, even when it might perfectly match their current, specific intention of joining on all shared-named columns. They opt for a verbose `ON` clause listing all common columns. While this explicit `ON` clause is generally safer and more readable regarding intent, the exercise highlights a scenario where a user *could* have used `NATURAL JOIN` for conciseness but chose not to, perhaps due to a blanket "never use NATURAL JOIN" rule. The "inefficiency" here is in terms of code verbosity and potentially missing out on a concise feature *when it's appropriately understood and the risks managed* (e.g., in ad-hoc queries or tightly controlled schemas). The general advice to prefer explicit joins (ON/USING) remains sound for maintainability.

1.3.3 Exercise 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`.

Solution: Inefficient N+1 Query Approach (Conceptual):

```
-- Step 1: Fetch all employees
-- Pseudocode: employees_list = SELECT employee_id, first_name, last_name, manager_id
    FROM employees;

-- Step 2: For each employee, if manager_id is present, fetch manager's name
-- Pseudocode:
-- FOR employee IN employees_list:
--   PRINT employee.first_name, employee.last_name
--   IF employee.manager_id IS NOT NULL THEN
--     manager_details = SELECT first_name, last_name FROM employees WHERE employee_id =
   employee.manager_id; -- Another query
--     PRINT "Managed by:", manager_details.first_name, manager_details.last_name
--   ELSE
--     PRINT "No manager"
--   END IF
```

This would result in 1 query to get all employees, and then up to N additional queries (where N is the number of employees with managers) to get each manager's details.

Efficient `SELF JOIN` Solution:

```sql
SELECT
    e.first_name || ' ' || e.last_name AS employee_name,
    COALESCE(m.first_name || ' ' || m.last_name, 'No Manager') AS
   manager_name
FROM
    employees e
LEFT JOIN
    employees m ON e.manager_id = m.employee_id
ORDER BY
    e.last_name, e.first_name;
```

Explanation: The N+1 query pattern is a common anti-pattern that leads to significant database overhead due to many small, repeated queries. It's often "easier" to think about procedurally but performs poorly. A `SELF JOIN` accomplishes the same task in a single, more efficient query by letting the database handle the relationships and data retrieval in one go. This reduces network latency and database processing time.

1.3.4 Exercise 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`.

Solution: Common `ON` clause (perfectly fine, but more verbose):

```sql
SELECT
    e.first_name,
    e.last_name,
    d.department_name
FROM
    employees e
INNER JOIN
    departments d ON e.department_id = d.department_id;
```

Concise `USING` clause solution:

```sql
SELECT
    e.first_name,
    e.last_name,
    d.department_name
FROM
    employees e
INNER JOIN
    departments d USING (department_id);
```

Explanation: Consistently writing out the full `ON e.col = d.col` when `USING(col)` is applicable is not "highly inefficient" in terms of query execution performance. However, it is less "code-efficient" or "typing-efficient." The `USING` clause offers:

1. **Conciseness:** It's shorter to write and read for simple equi-joins on identically named columns.

2. **De-duplication of Join Columns in `SELECT *`:** If `SELECT *` is used (though generally not recommended in production code), `USING` typically outputs the join column only once, whereas `ON` would output it from both tables (e.g., `department_id` from `employees` and `department_id` from `departments`).

While the `ON` clause is more explicit and generally preferred for its clarity and flexibility, deliberately avoiding `USING` in all cases where it *could* simplify the query (and the join is indeed on a single, identically named column) means sacrificing some brevity. This is a minor point compared to performance inefficiencies but relates to the "easier, basic, common" aspect where the slightly more verbose `ON` is often the default even if `USING` is perfectly suitable.

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

1.4.1 Exercise 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:

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

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

3. Manages at least 2 employees.

4. 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.

Solution:

```sql
WITH ManagerDirectReports AS (
    -- Step 1: Identify all employees and their direct managers
    SELECT
        m.employee_id AS manager_id,
        m.first_name AS manager_first_name,
        m.last_name AS manager_last_name,
        m.job_title AS manager_job_title,
        m.hire_date AS manager_hire_date,
        m.department_id AS manager_department_id,
        e.employee_id AS report_id,
        e.salary AS report_salary
    FROM
        employees m -- Potential managers
    INNER JOIN
        employees e ON m.employee_id = e.manager_id -- e are direct
   reports of m
),
ManagerStats AS (
```

```sql
    -- Step 2: Calculate stats for each manager (num reports, avg salary of reports)
    SELECT
        manager_id,
        manager_first_name,
        manager_last_name,
        manager_job_title,
        manager_hire_date,
        manager_department_id,
        COUNT(report_id) AS num_direct_reports,
        AVG(report_salary) AS avg_reports_salary
    FROM
        ManagerDirectReports
    GROUP BY
        manager_id, manager_first_name, manager_last_name, manager_job_title, manager_hire_date, manager_department_id
)
-- Step 3: Filter managers based on criteria and join with department/location info
SELECT
    ms.manager_first_name || ' ' || ms.manager_last_name AS manager_name,
    ms.manager_job_title,
    d.department_name,
    l.city AS department_city,
    ms.num_direct_reports,
    TO_CHAR(ms.avg_reports_salary, 'FM999999.00') AS avg_reports_salary_formatted
FROM
    ManagerStats ms
INNER JOIN
    departments d ON ms.manager_department_id = d.department_id
INNER JOIN
    locations l ON d.location_id = l.location_id
WHERE
    l.country = 'USA'                        -- Condition 1 (department in USA)
    AND ms.manager_hire_date <= '2020-01-01'  -- Condition 2 (manager hired on/before date)
    AND ms.num_direct_reports >= 2            -- Condition 3 (manages at least 2 employees)
    AND ms.avg_reports_salary > 65000         -- Condition 4 (avg salary of reports > 65k)
ORDER BY
    ms.manager_last_name;

-- Commentary on NATURAL JOIN and CROSS JOIN:
-- NATURAL JOIN:
--   - Between `employees` and `departments`: If `department_id` was the *only* common column and its meaning was identical for the join, `NATURAL JOIN` could be used. However, `employees` also has `manager_id` which could conflict if `departments` coincidentally had a `manager_id` column for a different purpose. `USING(department_id)` is safer and clearer.
--   - Between `departments` and `projects`: Both have `department_id`. If the intent is to link projects to their owning departments, `NATURAL JOIN` would work if `department_id` is the sole shared column name intended for the join. Risk: if another column, say `
```

```
    budget_code`, were added to both with the same name but different
    meanings, `NATURAL JOIN` would incorrectly try to use it. `USING(
    department_id)` or `ON d.department_id = p.department_id` is more
    robust.
-- CROSS JOIN:
--    - Not directly useful for the primary logic of this problem, which
      involves specific relationships (manager-report, employee-
    department, department-location). A `CROSS JOIN` would create
    combinations that are not meaningful for these directed
    relationships (e.g., every manager with every department,
    irrespective of their actual assignment). It could be used for
    generating all possible manager-project pairings if that was a
    requirement, but not for this problem's specific filtering.
```

Explanation of Hardcore Problem Solution:

1. `ManagerDirectReports` **CTE (SELF JOIN):** This CTE uses a `SELF JOIN` on the `employees` table (aliased as `m` for managers and `e` for employees/reports) to link each manager to their direct reports. It collects essential details for both. This is a core use of `SELF JOIN` for hierarchical data.

2. `ManagerStats` **CTE (Aggregation):** This CTE takes the results from `ManagerDirectReports` and groups by manager to calculate `COUNT(report_id)` (number of direct reports) and `AVG(report_salary)` (average salary of those reports). This uses standard SQL aggregation (`GROUP BY`, `COUNT`, `AVG`).

3. **Final** `SELECT` **Statement (JOINs and Filtering):**

   - It joins `ManagerStats` with `departments` (using `ON ms.manager_department_id = d.department_id` - `USING(department_id)` would also work here if `manager_department_i` was named `department_id` in the CTE) and then with `locations` (`ON d.location_id = l.location_id`). These are standard `INNER JOINs`.
   - The `WHERE` clause applies all the specified conditions for a "High-Impact Manager":
     - `l.country = 'USA'` (Basic SQL `WHERE`, String comparison)
     - `ms.manager_hire_date <= '2020-01-01'` (Date comparison)
     - `ms.num_direct_reports >= 2` (Arithmetic comparison)
     - `ms.avg_reports_salary > 65000` (Arithmetic comparison)
   - `TO_CHAR` is used for formatting the average salary (PostgreSQL specific, similar to `FORMAT` in other SQL dialects).
   - `ORDER BY` clause for sorting.
   - The problem covers concepts like CTEs (for readability and stepwise logic), SELF JOINs, standard JOINs, aggregation functions, date functions/comparisons, string comparisons, arithmetic, and conditional logic in `WHERE`.

This problem effectively utilizes `SELF JOIN` for its core logic and combines it with various other SQL concepts from Basic and Intermediate levels, as required. The commentary addresses the other join types (`NATURAL JOIN`, `CROSS JOIN`) from the "Joins" section of Complementary SQL.

2 Aggregators (COUNT(DISTINCT), FILTER clause)

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

2.1.1 Exercise 5.1 (COUNT(DISTINCT column) - Meaning & Advantage)

Problem: The sales department wants to know how many unique customers have made purchases from the `sales_data` table.

Solution:

```sql
SELECT COUNT(DISTINCT customer_id_text) AS unique_customer_count
FROM sales_data;
```

Explanation: `COUNT(DISTINCT customer_id_text)` calculates the number of unique non-null values in the `customer_id_text` column.

- **Meaning:** It counts each distinct customer identifier only once, regardless of how many purchases they made.
- **Value:** Provides an accurate count of the customer base that has engaged in transactions.
- **Relation to `COUNT(*)` or `COUNT(column)`:** `COUNT(*)` would count all rows (total sales transactions). `COUNT(customer_id_text)` would count all non-null customer IDs, including duplicates. `COUNT(DISTINCT ...)` is specifically for unique counts.
- **Advantage:** Essential for metrics like "unique visitors," "unique customers," etc., preventing overcounting when an entity appears multiple times.

2.1.2 Exercise 5.2 (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.

Solution:

```sql
SELECT
    COUNT(*) AS total_sales_transactions,
    COUNT(*) FILTER (WHERE region = 'Europe') AS
   european_sales_transactions
FROM sales_data;
```

Explanation: The `FILTER (WHERE condition)` clause is used with aggregate functions to perform aggregation only on rows that satisfy the `condition`.

- **Meaning:** `COUNT(*) FILTER (WHERE region = 'Europe')` counts only those rows where the `region` column is 'Europe'.
- **Value:** Allows for multiple conditional aggregations within a single query and a single `GROUP BY` (if groups were used).
- **Relation to `CASE` statements:** An older way to achieve this is `COUNT(CASE WHEN region = 'Europe' THEN 1 END)`.

- **Advantage:** The `FILTER` clause is often more readable and can be more efficient than the `CASE` statement approach for conditional aggregation, especially with multiple conditions. It's standard SQL, making queries cleaner.

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

2.2.1 Exercise 6.1 (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(*)`?

Solution:

```sql
-- Example query that might be slow on a huge table without proper indexing:
SELECT COUNT(DISTINCT very_large_column_with_many_unique_values)
FROM extremely_large_table;
```

Explanation:

- **Disadvantage:** `COUNT(DISTINCT column)` can be significantly slower than `COUNT(*)` or `COUNT(column_with_few_nulls)`.
  - To count distinct values, the database typically needs to identify all unique values first. This often involves sorting the distinct values or using a hash-based aggregation strategy to keep track of unique values encountered. Both operations can be memory and CPU intensive, especially if the number of distinct values (cardinality) is high and the dataset is large.
  - If the column is not indexed, the database might need to perform a full table scan and then sort/hash a large amount of data. Even with an index, if the cardinality is very high, processing can be demanding.
  - `COUNT(*)` simply counts rows, which is often a faster operation, especially if the table has a compact structure or metadata about row counts is readily available.

2.2.2 Exercise 6.2 (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?

Solution:

```sql
-- FILTER clause (PostgreSQL and modern SQL standard)
SELECT
    SUM(quantity_sold) FILTER (WHERE region = 'North America') AS na_total_quantity,
    SUM(quantity_sold) FILTER (WHERE region = 'Europe') AS eu_total_quantity
FROM sales_data;

```

```sql
-- Alternative using CASE (more portable to older RDBMS or those without FILTER)
SELECT
    SUM(CASE WHEN region = 'North America' THEN quantity_sold ELSE 0 END) AS na_total_quantity,
    SUM(CASE WHEN region = 'Europe' THEN quantity_sold ELSE 0 END) AS eu_total_quantity
FROM sales_data;
```

Explanation:

- **Disadvantage (Portability/Availability):** The main disadvantage isn't in its concept but in its historical availability.
  1. **RDBMS Support:** Not all RDBMS versions support the `FILTER` clause, or they might have implemented it later than other features. If you need to write SQL that runs on older database versions or a variety of database systems, some of which lack `FILTER` support, your query won't be portable.
  2. **Alternative Required:** You would have to resort to using `CASE` expressions inside aggregate functions (e.g., `SUM(CASE WHEN condition THEN value ELSE NULL END)` or `COUNT(CASE WHEN condition THEN 1 END)`). While functionally equivalent, this can make the query more verbose and sometimes less readable than the `FILTER` clause.
- The `CASE` approach is generally well-supported across most SQL databases, making it a more universal solution for conditional aggregation when `FILTER` is not an option.

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

2.3.1 Exercise 7.1 (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.

Solution: Inefficient/Less Direct Approach:

```sql
SELECT COUNT(*) AS unique_products_sold
FROM (
    SELECT DISTINCT product_id
    FROM sales_data
) AS distinct_products;
```

Efficient `COUNT(DISTINCT)` Solution:

```sql
SELECT COUNT(DISTINCT product_id) AS unique_products_sold
FROM sales_data;
```

Explanation: While modern query optimizers might handle both forms similarly, the `SELECT COUNT(*) FROM (SELECT DISTINCT ...)` approach is more verbose and explicitly tells the database to first materialize the set of distinct product IDs and then count them. COUNT(DISTINCT product_id) is a direct instruction to perform this specific aggregation. For very large datasets or older/simpler database systems, the direct COUNT(DISTINCT) might allow for more specialized internal optimizations (like stream-based distinct counting) that might not be as readily applied to the subquery form. The direct form is also more concise and clearly states the intent. Another highly inefficient method would be fetching all product_id values to the application layer and then using a Set data structure to count uniques, which involves unnecessary data transfer.

2.3.2 Exercise 7.2 (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.

Solution: Inefficient Approach 1: Multiple Queries (Conceptual)

```sql
-- Query 1
-- SELECT COUNT(*) AS total_sales FROM sales_data;
-- Query 2
-- SELECT COUNT(*) AS na_sales FROM sales_data WHERE region = 'North America';
-- Query 3
-- SELECT COUNT(*) AS paypal_sales FROM sales_data WHERE payment_method = 'PayPal';
-- (Results then combined manually or in application code)
```

Alternative Approach 2: Using SUM(CASE...) (More verbose for counts)

```sql
SELECT
    COUNT(*) AS total_sales,
    SUM(CASE WHEN region = 'North America' THEN 1 ELSE 0 END) AS na_sales_count,
    SUM(CASE WHEN payment_method = 'PayPal' THEN 1 ELSE 0 END) AS paypal_sales_count
FROM sales_data;
-- Note: COUNT(CASE WHEN ... THEN 1 END) is often preferred for counts over SUM to avoid issues if the column could be non-numeric.
-- Using COUNT(CASE ...):
SELECT
    COUNT(*) AS total_sales,
    COUNT(CASE WHEN region = 'North America' THEN 1 END) AS na_sales_count, -- NULLs from ELSE are not counted by COUNT(expression)
    COUNT(CASE WHEN payment_method = 'PayPal' THEN 1 END) AS paypal_sales_count
FROM sales_data;
```

Efficient FILTER Clause Solution:

```sql
SELECT
    COUNT(*) AS total_sales,
    COUNT(*) FILTER (WHERE region = 'North America') AS na_sales_count,
    COUNT(*) FILTER (WHERE payment_method = 'PayPal') AS paypal_sales_count
FROM sales_data;
```

Explanation:

- **Multiple Queries:** This is highly inefficient due to repeated full table scans (unless indexed appropriately and queries are very simple) and the overhead of multiple round trips to the database.

- **`SUM(CASE...)` or `COUNT(CASE...)`:** This is a valid and common way to do conditional aggregation. However, for straightforward conditional counts, the `FILTER` clause is arguably more declarative and readable (`COUNT(*) FILTER (WHERE condition)` directly expresses "count rows where condition is true"). The `CASE` syntax is more general purpose but can make simple conditional counts slightly more verbose. For complex conditions or when `FILTER` isn't available, `CASE` is the go-to. Using `FILTER` when available can lead to cleaner and sometimes more optimized queries. The "inefficiency" of `CASE` here is minor and more about readability/expressiveness for this specific task compared to `FILTER`.

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

2.4.1 Exercise 8.1 (Aggregators - Hardcore Problem)

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

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

2. `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.

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

4. `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`).

5. `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`).

6. `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.

Solution:

```sql
WITH CategorySalesAggregation AS (
    SELECT
        c.category_id,
        c.category_name,
        SUM(sd.quantity_sold * sd.unit_price_at_sale * (1 - sd.
    discount_percentage)) AS raw_total_revenue,
        COUNT(DISTINCT sd.customer_id_text) AS unique_customers_count,
        COUNT(*) FILTER (
            WHERE e.performance_rating = 5
        ) AS high_perf_employee_sales_count,
        COUNT(*) FILTER (
            WHERE (sd.quantity_sold * sd.unit_price_at_sale) > 200
              AND sd.region = 'North America'
              AND sd.payment_method = 'Credit Card'
        ) AS high_value_cc_sales_usa_count
    FROM
        sales_data sd
    JOIN
        products p ON sd.product_id = p.product_id
    JOIN
        categories c ON p.category_id = c.category_id
    JOIN
        employees e ON sd.employee_id = e.employee_id
    GROUP BY
        c.category_id, c.category_name
),
RankedCategories AS (
    SELECT
        category_name,
        raw_total_revenue,
        TO_CHAR(raw_total_revenue, 'FM9,999,990.00') AS total_revenue,
    -- Formatted revenue
        unique_customers_count,
        high_perf_employee_sales_count,
```

```
        high_value_cc_sales_usa_count,
        DENSE_RANK() OVER (ORDER BY raw_total_revenue DESC) AS category_revenue_rank
    FROM
        CategorySalesAggregation
    WHERE
        high_perf_employee_sales_count >= 1  -- Apply HAVING conditions here or in final SELECT
        AND unique_customers_count > 2
)
SELECT
    category_name,
    total_revenue,
    unique_customers_count,
    high_perf_employee_sales_count,
    high_value_cc_sales_usa_count,
    category_revenue_rank
FROM
    RankedCategories
ORDER BY
    category_revenue_rank ASC, category_name ASC;
```

Explanation of Hardcore Problem Solution:

1. `CategorySalesAggregation` **CTE:**
   - **Joins:** It starts by joining `sales_data` (fact table) with `products`, `categories` (dimension tables to get category info), and `employees` (to get performance rating).
   - `GROUP BY c.category_id, c.category_name`**:** Aggregations are performed per category.
   - `SUM(...)` **AS** `raw_total_revenue`**:** Calculates total revenue using arithmetic.
   - `COUNT(DISTINCT sd.customer_id_text)`**:** Calculates unique customers for the category. This is a direct use of `COUNT(DISTINCT)`.
   - `COUNT(*) FILTER (WHERE e.performance_rating = 5)`**:** Counts sales by high-performance employees using the `FILTER` clause. This showcases conditional aggregation.
   - `COUNT(*) FILTER (WHERE (sd.quantity_sold * sd.unit_price_at_sale) > 200 ...)`**:** Counts high-value credit card sales in the USA using `FILTER` with multiple conditions. This demonstrates a more complex conditional aggregation.
2. `RankedCategories` **CTE:**
   - This CTE takes data from `CategorySalesAggregation`.
   - `TO_CHAR(...)` **AS** `total_revenue`**:** Formats the raw revenue into a currency string.
   - `DENSE_RANK() OVER (ORDER BY raw_total_revenue DESC)`**:** Assigns a rank to each category based on its `raw_total_revenue`. This uses a Ranking Window Function as allowed.

- `WHERE high_perf_employee_sales_count >= 1 AND unique_customers_count > 2`: This applies the filtering criteria that would typically be in a `HAVING` clause if we weren't using window functions that are calculated after `GROUP BY/HAVING`. Placing it here filters before the final select or could be in the final select.

3. **Final `SELECT` Statement:**

   - Selects the required columns from the `RankedCategories` CTE.
   - `ORDER BY category_revenue_rank ASC, category_name ASC`: Sorts the final output as required.

This problem comprehensively uses `COUNT(DISTINCT)`, the `FILTER` clause for different conditional aggregations, and integrates them with JOINs, GROUP BY, standard aggregators, arithmetic, and Window Functions (Ranking), addressing all aspects of the hardcore problem requirements for "Aggregators".