/\*  
-- Employees Table  
CREATE TABLE employees (  
 employee\_id INT PRIMARY KEY,  
 first\_name VARCHAR(50),  
 last\_name VARCHAR(50),  
 full\_name VARCHAR(100), -- For CTE examples  
 salary DECIMAL(10,2),  
 department\_id INT,  
 manager\_id INT,  
 hire\_date DATE,  
 revenue\_generated DECIMAL(15,2)  
);  
  
-- Customers Table  
CREATE TABLE customers (  
 customer\_id INT PRIMARY KEY,  
 customer\_name VARCHAR(100),  
 registration\_date DATE,  
 region VARCHAR(50)  
);  
  
-- Orders Table  
CREATE TABLE orders (  
 order\_id INT PRIMARY KEY,  
 customer\_id INT,  
 employee\_id INT,  
 order\_date DATE,  
 order\_value DECIMAL(10,2),  
 product\_id INT  
);  
  
-- Products Table  
CREATE TABLE products (  
 product\_id INT PRIMARY KEY,  
 product\_name VARCHAR(100),  
 category VARCHAR(50),  
 price DECIMAL(10,2)  
);  
  
-- Sales Table  
CREATE TABLE sales (  
 sale\_id INT PRIMARY KEY,  
 employee\_id INT,  
 sale\_date DATE,  
 sale\_amount DECIMAL(10,2),  
 region VARCHAR(50)  
);  
  
-- Projects Table  
CREATE TABLE projects (  
 project\_id INT PRIMARY KEY,  
 employee\_id INT,  
 project\_name VARCHAR(100)  
);  
\*/  
  
-- =====================================================  
-- WINDOW FUNCTIONS (5 Questions)  
-- =====================================================  
  
-- 1. Compute the rolling average of sales for the last three months  
-- Approach: Use window function with ROWS BETWEEN for sliding window calculation  
-- Performance: Index on (employee\_id, sale\_date) for optimal performance  
  
-- PostgreSQL/SQL Server/Oracle Solution:  
WITH monthly\_sales AS (  
 SELECT  
 employee\_id,  
 *DATE\_TRUNC*('month', sale\_date) as sale\_month, -- PostgreSQL  
 -- DATEPART(YEAR, sale\_date) \* 100 + DATEPART(MONTH, sale\_date) as sale\_month, -- SQL Server  
 -- TRUNC(sale\_date, 'MM') as sale\_month, -- Oracle  
 *SUM*(sale\_amount) as monthly\_total  
 FROM sales  
 GROUP BY employee\_id, *DATE\_TRUNC*('month', sale\_date)  
)  
SELECT  
 employee\_id,  
 sale\_month,  
 monthly\_total,  
 *AVG*(monthly\_total) OVER (  
 PARTITION BY employee\_id  
 ORDER BY sale\_month  
 ROWS BETWEEN 2 PRECEDING AND CURRENT ROW  
 ) as rolling\_3\_month\_avg  
FROM monthly\_sales  
ORDER BY employee\_id, sale\_month;  
  
-- MySQL Alternative (no DATE\_TRUNC):  
-- SELECT  
-- employee\_id,  
-- YEAR(sale\_date) \* 100 + MONTH(sale\_date) as sale\_month,  
-- SUM(sale\_amount) as monthly\_total,  
-- AVG(SUM(sale\_amount)) OVER (  
-- PARTITION BY employee\_id  
-- ORDER BY YEAR(sale\_date) \* 100 + MONTH(sale\_date)  
-- ROWS BETWEEN 2 PRECEDING AND CURRENT ROW  
-- ) as rolling\_3\_month\_avg  
-- FROM sales  
-- GROUP BY employee\_id, YEAR(sale\_date), MONTH(sale\_date)  
-- ORDER BY employee\_id, sale\_month;  
  
-- Index Recommendation:  
-- CREATE INDEX idx\_sales\_emp\_date ON sales(employee\_id, sale\_date, sale\_amount);  
  
-- =====================================================  
  
-- 2. Rank employees uniquely based on their salaries in descending order  
-- Approach: ROW\_NUMBER() ensures unique ranking even for tied salaries  
-- Performance: Index on salary column for fast sorting  
  
SELECT  
 employee\_id,  
 first\_name,  
 last\_name,  
 salary,  
 *ROW\_NUMBER*() OVER (ORDER BY salary DESC) as unique\_rank,  
 *RANK*() OVER (ORDER BY salary DESC) as standard\_rank,  
 *DENSE\_RANK*() OVER (ORDER BY salary DESC) as dense\_rank  
FROM employees  
ORDER BY salary DESC;  
  
-- Alternative with tie-breaking by employee\_id:  
SELECT  
 employee\_id,  
 first\_name,  
 last\_name,  
 salary,  
 *ROW\_NUMBER*() OVER (ORDER BY salary DESC, employee\_id ASC) as unique\_rank\_with\_tiebreaker  
FROM employees  
ORDER BY salary DESC, employee\_id ASC;  
  
-- Index Recommendation:  
-- CREATE INDEX idx\_employees\_salary ON employees(salary DESC, employee\_id);  
  
-- =====================================================  
  
-- 3. Identify the earliest and latest purchase dates for each customer  
-- Approach: FIRST\_VALUE and LAST\_VALUE with proper frame specification  
-- Performance: Index on (customer\_id, order\_date) for optimal window function performance  
  
SELECT DISTINCT  
 customer\_id,  
 *FIRST\_VALUE*(order\_date) OVER (  
 PARTITION BY customer\_id  
 ORDER BY order\_date  
 ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING  
 ) as earliest\_purchase,  
 *LAST\_VALUE*(order\_date) OVER (  
 PARTITION BY customer\_id  
 ORDER BY order\_date  
 ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING  
 ) as latest\_purchase,  
 *COUNT*(\*) OVER (PARTITION BY customer\_id) as total\_orders  
FROM orders  
ORDER BY customer\_id;  
  
-- Alternative using MIN/MAX (often more efficient):  
SELECT  
 customer\_id,  
 *MIN*(order\_date) as earliest\_purchase,  
 *MAX*(order\_date) as latest\_purchase,  
 *COUNT*(\*) as total\_orders  
FROM orders  
GROUP BY customer\_id  
ORDER BY customer\_id;  
  
-- Index Recommendation:  
-- CREATE INDEX idx\_orders\_customer\_date ON orders(customer\_id, order\_date);  
  
-- =====================================================  
  
-- 4. Find the second highest salary in each department using window functions  
-- Approach: DENSE\_RANK to handle tied salaries, filter for rank = 2  
-- Performance: Composite index on (department\_id, salary) for efficient sorting  
  
WITH ranked\_salaries AS (  
 SELECT  
 employee\_id,  
 first\_name,  
 last\_name,  
 department\_id,  
 salary,  
 *DENSE\_RANK*() OVER (  
 PARTITION BY department\_id  
 ORDER BY salary DESC  
 ) as salary\_rank  
 FROM employees  
)  
SELECT  
 employee\_id,  
 first\_name,  
 last\_name,  
 department\_id,  
 salary  
FROM ranked\_salaries  
WHERE salary\_rank = 2  
ORDER BY department\_id, salary DESC;  
  
-- Alternative using OFFSET for single result per department:  
SELECT DISTINCT  
 department\_id,  
 *NTH\_VALUE*(salary, 2) OVER (  
 PARTITION BY department\_id  
 ORDER BY salary DESC  
 ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING  
 ) as second\_highest\_salary  
FROM employees  
WHERE *NTH\_VALUE*(salary, 2) OVER (  
 PARTITION BY department\_id  
 ORDER BY salary DESC  
 ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING  
) IS NOT NULL  
ORDER BY department\_id;  
  
-- Index Recommendation:  
-- CREATE INDEX idx\_employees\_dept\_salary ON employees(department\_id, salary DESC);  
  
-- =====================================================  
  
-- 5. Calculate the percentage contribution of each employee to the company's total revenue  
-- Approach: Use SUM() OVER() for total revenue, calculate percentage  
-- Performance: Consider materialized view for frequently accessed company totals  
  
SELECT  
 employee\_id,  
 first\_name,  
 last\_name,  
 revenue\_generated,  
 *SUM*(revenue\_generated) OVER () as total\_company\_revenue,  
 *ROUND*(  
 (revenue\_generated / *SUM*(revenue\_generated) OVER ()) \* 100,  
 2  
 ) as percentage\_contribution,  
 -- Running percentage for cumulative analysis  
 *ROUND*(  
 (*SUM*(revenue\_generated) OVER (ORDER BY revenue\_generated DESC) /  
 SUM(revenue\_generated) OVER ()) \* 100,  
 2  
 ) as cumulative\_percentage  
FROM employees  
WHERE revenue\_generated > 0  
ORDER BY revenue\_generated DESC;  
  
-- Alternative with CASE for zero division protection:  
SELECT  
 employee\_id,  
 first\_name,  
 last\_name,  
 revenue\_generated,  
 CASE  
 WHEN *SUM*(revenue\_generated) OVER () = 0 THEN 0  
 ELSE *ROUND*((revenue\_generated / *SUM*(revenue\_generated) OVER ()) \* 100, 2)  
 END as percentage\_contribution  
FROM employees  
ORDER BY revenue\_generated DESC;  
  
-- =====================================================  
-- COMMON TABLE EXPRESSIONS (CTEs) (5 Questions)  
-- =====================================================  
  
-- 1. Use a CTE to separate full names into first and last names  
-- Approach: String manipulation functions to parse full\_name column  
-- Cross-platform note: String functions vary across databases  
  
-- PostgreSQL Solution:  
WITH name\_parser AS (  
 SELECT  
 employee\_id,  
 full\_name,  
 *SPLIT\_PART*(full\_name, ' ', 1) as parsed\_first\_name,  
 CASE  
 WHEN *POSITION*(' ' IN full\_name) > 0  
 THEN *SUBSTRING*(full\_name FROM *POSITION*(' ' IN full\_name) + 1)  
 ELSE ''  
 END as parsed\_last\_name  
 FROM employees  
 WHERE full\_name IS NOT NULL  
)  
SELECT  
 employee\_id,  
 full\_name,  
 parsed\_first\_name,  
 parsed\_last\_name,  
 -- Validation: Check if parsing matches existing data  
 CASE  
 WHEN first\_name = parsed\_first\_name AND last\_name = parsed\_last\_name  
 THEN 'Match'  
 ELSE 'Mismatch'  
 END as validation\_status  
FROM name\_parser np  
JOIN employees e USING (employee\_id)  
ORDER BY employee\_id;  
  
-- SQL Server Alternative:  
-- WITH name\_parser AS (  
-- SELECT  
-- employee\_id,  
-- full\_name,  
-- LEFT(full\_name, CHARINDEX(' ', full\_name + ' ') - 1) as parsed\_first\_name,  
-- LTRIM(SUBSTRING(full\_name, CHARINDEX(' ', full\_name + ' '), LEN(full\_name))) as parsed\_last\_name  
-- FROM employees  
-- WHERE full\_name IS NOT NULL  
-- )  
  
-- MySQL Alternative:  
-- WITH name\_parser AS (  
-- SELECT  
-- employee\_id,  
-- full\_name,  
-- SUBSTRING\_INDEX(full\_name, ' ', 1) as parsed\_first\_name,  
-- SUBSTRING\_INDEX(full\_name, ' ', -1) as parsed\_last\_name  
-- FROM employees  
-- WHERE full\_name IS NOT NULL  
-- )  
  
-- =====================================================  
  
-- 2. Write a CTE to determine the longest streak of consecutive sales by an employee  
-- Approach: Use ROW\_NUMBER and grouping to identify consecutive sequences  
-- Performance: Index on (employee\_id, sale\_date) essential for performance  
  
WITH sales\_with\_sequence AS (  
 -- Add row numbers to identify gaps in consecutive dates  
 SELECT  
 employee\_id,  
 sale\_date,  
 sale\_amount,  
 *ROW\_NUMBER*() OVER (PARTITION BY employee\_id ORDER BY sale\_date) as rn,  
 sale\_date - INTERVAL '1 day' \* ROW\_NUMBER() OVER (PARTITION BY employee\_id ORDER BY sale\_date) as group\_date  
 FROM sales  
),  
consecutive\_groups AS (  
 -- Group consecutive sales together  
 SELECT  
 employee\_id,  
 group\_date,  
 COUNT(\*) as streak\_length,  
 MIN(sale\_date) as streak\_start,  
 MAX(sale\_date) as streak\_end,  
 SUM(sale\_amount) as streak\_total\_sales  
 FROM sales\_with\_sequence  
 GROUP BY employee\_id, group\_date  
),  
longest\_streaks AS (  
 -- Find the longest streak for each employee  
 SELECT  
 employee\_id,  
 streak\_length,  
 streak\_start,  
 streak\_end,  
 streak\_total\_sales,  
 ROW\_NUMBER() OVER (PARTITION BY employee\_id ORDER BY streak\_length DESC, streak\_total\_sales DESC) as rn  
 FROM consecutive\_groups  
)  
SELECT  
 ls.employee\_id,  
 e.first\_name,  
 e.last\_name,  
 ls.streak\_length as longest\_consecutive\_days,  
 ls.streak\_start,  
 ls.streak\_end,  
 ls.streak\_total\_sales  
FROM longest\_streaks ls  
JOIN employees e ON ls.employee\_id = e.employee\_id  
WHERE ls.rn = 1  
ORDER BY ls.streak\_length DESC;  
  
-- Index Recommendation:  
-- CREATE INDEX idx\_sales\_emp\_date\_amount ON sales(employee\_id, sale\_date, sale\_amount);  
  
-- =====================================================  
  
-- 3. Generate a sequence of Fibonacci numbers up to a specific value using a recursive CTE  
-- Approach: Recursive CTE with termination condition  
-- Performance: Limit recursion depth to prevent infinite loops  
  
WITH RECURSIVE fibonacci\_sequence AS (  
 -- Base case: First two Fibonacci numbers  
 SELECT  
 1 as position,  
 0 as fib\_number,  
 1 as next\_fib  
  
 UNION ALL  
  
 -- Recursive case: Generate next Fibonacci number  
 SELECT  
 position + 1,  
 next\_fib,  
 fib\_number + next\_fib  
 FROM fibonacci\_sequence  
 WHERE next\_fib <= 1000 -- Limit: generate up to 1000  
 AND position < 50 -- Safety: prevent excessive recursion  
)  
SELECT  
 position,  
 fib\_number,  
 -- Additional calculations  
 CASE  
 WHEN position > 1  
 THEN *ROUND*(fib\_number::NUMERIC / LAG(fib\_number) OVER (ORDER BY position), 6)  
 ELSE NULL  
 END as golden\_ratio\_approximation  
FROM fibonacci\_sequence  
WHERE fib\_number <= 1000  
ORDER BY position;  
  
-- Alternative with parameterized limit:  
-- Replace 1000 with @max\_value parameter in stored procedure  
  
-- Cross-platform note:  
-- - PostgreSQL: Works as shown  
-- - SQL Server: Use CAST instead of ::NUMERIC  
-- - MySQL: Supported in 8.0+  
-- - Oracle: Use CONNECT BY for recursive behavior  
  
-- =====================================================  
  
-- 4. Use a CTE to detect and list duplicate entries in a table  
-- Approach: Window functions to count occurrences and identify duplicates  
-- Performance: Index on columns being checked for duplicates  
  
WITH duplicate\_analysis AS (  
 SELECT  
 customer\_id,  
 customer\_name,  
 email,  
 phone,  
 *COUNT*(\*) OVER (PARTITION BY customer\_name, email) as name\_email\_count,  
 *COUNT*(\*) OVER (PARTITION BY phone) as phone\_count,  
 *COUNT*(\*) OVER (PARTITION BY email) as email\_count,  
 *ROW\_NUMBER*() OVER (PARTITION BY customer\_name, email ORDER BY customer\_id) as duplicate\_rank  
 FROM customers  
),  
duplicate\_summary AS (  
 SELECT  
 customer\_id,  
 customer\_name,  
 email,  
 phone,  
 name\_email\_count,  
 phone\_count,  
 email\_count,  
 duplicate\_rank,  
 -- Categorize type of duplicate  
 CASE  
 WHEN name\_email\_count > 1 AND phone\_count > 1 THEN 'Complete Duplicate'  
 WHEN name\_email\_count > 1 THEN 'Name-Email Duplicate'  
 WHEN phone\_count > 1 THEN 'Phone Duplicate'  
 WHEN email\_count > 1 THEN 'Email Duplicate'  
 ELSE 'No Duplicate'  
 END as duplicate\_type  
 FROM duplicate\_analysis  
)  
SELECT  
 customer\_id,  
 customer\_name,  
 email,  
 phone,  
 duplicate\_type,  
 name\_email\_count,  
 duplicate\_rank,  
 -- Recommendation for cleanup  
 CASE  
 WHEN duplicate\_rank = 1 THEN 'KEEP'  
 ELSE 'REVIEW FOR DELETION'  
 END as recommendation  
FROM duplicate\_summary  
WHERE name\_email\_count > 1 OR phone\_count > 1 OR email\_count > 1  
ORDER BY duplicate\_type, customer\_name, duplicate\_rank;  
  
-- Cleanup query to remove duplicates (keep first occurrence):  
-- DELETE FROM customers  
-- WHERE customer\_id IN (  
-- SELECT customer\_id FROM duplicate\_summary  
-- WHERE duplicate\_rank > 1 AND duplicate\_type = 'Complete Duplicate'  
-- );  
  
-- Index Recommendations:  
-- CREATE INDEX idx\_customers\_name\_email ON customers(customer\_name, email);  
-- CREATE INDEX idx\_customers\_phone ON customers(phone);  
-- CREATE INDEX idx\_customers\_email ON customers(email);  
  
-- =====================================================  
  
-- 5. Calculate total sales per category and filter out categories with sales below a specific threshold using a CTE  
-- Approach: Aggregate sales by category, then filter using CTE  
-- Performance: Index on (category, price) for efficient aggregation  
  
WITH category\_sales AS (  
 SELECT  
 p.category,  
 *COUNT*(DISTINCT o.order\_id) as total\_orders,  
 *COUNT*(DISTINCT o.customer\_id) as unique\_customers,  
 *SUM*(o.order\_value) as total\_sales,  
 *AVG*(o.order\_value) as avg\_order\_value,  
 *MIN*(o.order\_date) as first\_sale\_date,  
 *MAX*(o.order\_date) as last\_sale\_date  
 FROM orders o  
 JOIN products p ON o.product\_id = p.product\_id  
 GROUP BY p.category  
),  
sales\_ranking AS (  
 SELECT  
 \*,  
 *RANK*() OVER (ORDER BY total\_sales DESC) as sales\_rank,  
 *ROUND*((total\_sales / *SUM*(total\_sales) OVER ()) \* 100, 2) as market\_share\_pct  
 FROM category\_sales  
)  
SELECT  
 category,  
 total\_orders,  
 unique\_customers,  
 total\_sales,  
 avg\_order\_value,  
 sales\_rank,  
 market\_share\_pct,  
 first\_sale\_date,  
 last\_sale\_date,  
 -- Performance indicators  
 *ROUND*(total\_sales / total\_orders, 2) as revenue\_per\_order,  
 *ROUND*(total\_sales / unique\_customers, 2) as revenue\_per\_customer  
FROM sales\_ranking  
WHERE total\_sales >= 50000 -- Threshold: $50,000 minimum sales  
ORDER BY total\_sales DESC;  
  
-- Dynamic threshold using parameters:  
-- DECLARE @sales\_threshold DECIMAL(15,2) = 50000;  
-- Add: WHERE total\_sales >= @sales\_threshold  
  
-- Index Recommendations:  
-- CREATE INDEX idx\_orders\_product\_value ON orders(product\_id, order\_value, order\_date);  
-- CREATE INDEX idx\_products\_category ON products(category, product\_id);  
  
-- =====================================================  
-- JOINS (5 Questions)  
-- =====================================================  
  
-- 1. List all customers, highlighting who placed orders and who didn't (Full Outer Join)  
-- Approach: FULL OUTER JOIN to show all customers and all orders  
-- Cross-platform note: MySQL doesn't support FULL OUTER JOIN directly  
  
-- PostgreSQL/SQL Server/Oracle Solution:  
SELECT  
 *COALESCE*(c.customer\_id, o.customer\_id) as customer\_id,  
 c.customer\_name,  
 c.registration\_date,  
 c.region,  
 *COUNT*(o.order\_id) as total\_orders,  
 *COALESCE*(*SUM*(o.order\_value), 0) as total\_order\_value,  
 *MAX*(o.order\_date) as last\_order\_date,  
 CASE  
 WHEN c.customer\_id IS NULL THEN 'Order without customer record'  
 WHEN o.customer\_id IS NULL THEN 'Customer with no orders'  
 ELSE 'Active customer'  
 END as customer\_status  
FROM customers c  
FULL OUTER JOIN orders o ON c.customer\_id = o.customer\_id  
GROUP BY  
 *COALESCE*(c.customer\_id, o.customer\_id),  
 c.customer\_name,  
 c.registration\_date,  
 c.region  
ORDER BY customer\_status, total\_order\_value DESC;  
  
-- MySQL Alternative (using UNION):  
-- SELECT  
-- c.customer\_id,  
-- c.customer\_name,  
-- c.registration\_date,  
-- c.region,  
-- COUNT(o.order\_id) as total\_orders,  
-- COALESCE(SUM(o.order\_value), 0) as total\_order\_value,  
-- 'Customer' as record\_type  
-- FROM customers c  
-- LEFT JOIN orders o ON c.customer\_id = o.customer\_id  
-- GROUP BY c.customer\_id, c.customer\_name, c.registration\_date, c.region  
--  
-- UNION ALL  
--  
-- SELECT  
-- o.customer\_id,  
-- 'UNKNOWN' as customer\_name,  
-- NULL as registration\_date,  
-- NULL as region,  
-- COUNT(o.order\_id) as total\_orders,  
-- SUM(o.order\_value) as total\_order\_value,  
-- 'Orphaned Order' as record\_type  
-- FROM orders o  
-- LEFT JOIN customers c ON o.customer\_id = c.customer\_id  
-- WHERE c.customer\_id IS NULL  
-- GROUP BY o.customer\_id;  
  
-- =====================================================  
  
-- 2. Identify employees assigned to more than one project using a self-join  
-- Approach: Self-join on projects table to find employees with multiple projects  
-- Performance: Index on employee\_id for efficient self-join  
  
SELECT DISTINCT  
 p1.employee\_id,  
 e.first\_name,  
 e.last\_name,  
 *COUNT*(DISTINCT p1.project\_id) as total\_projects,  
 *STRING\_AGG*(p1.project\_name, ', ' ORDER BY p1.project\_name) as project\_list  
FROM projects p1  
JOIN projects p2 ON p1.employee\_id = p2.employee\_id  
 AND p1.project\_id != p2.project\_id  
JOIN employees e ON p1.employee\_id = e.employee\_id  
GROUP BY p1.employee\_id, e.first\_name, e.last\_name  
HAVING COUNT(DISTINCT p1.project\_id) > 1  
ORDER BY total\_projects DESC, e.last\_name;  
  
-- Alternative using window function (often more efficient):  
WITH project\_counts AS (  
 SELECT  
 employee\_id,  
 project\_id,  
 project\_name,  
 *COUNT*(\*) OVER (PARTITION BY employee\_id) as project\_count  
 FROM projects  
)  
SELECT  
 pc.employee\_id,  
 e.first\_name,  
 e.last\_name,  
 pc.project\_count as total\_projects,  
 *STRING\_AGG*(pc.project\_name, ', ' ORDER BY pc.project\_name) as project\_list  
FROM project\_counts pc  
JOIN employees e ON pc.employee\_id = e.employee\_id  
WHERE pc.project\_count > 1  
GROUP BY pc.employee\_id, e.first\_name, e.last\_name, pc.project\_count  
ORDER BY pc.project\_count DESC, e.last\_name;  
  
-- Cross-platform STRING\_AGG alternatives:  
-- SQL Server: STRING\_AGG(project\_name, ', ')  
-- MySQL: GROUP\_CONCAT(project\_name ORDER BY project\_name SEPARATOR ', ')  
-- Oracle: LISTAGG(project\_name, ', ') WITHIN GROUP (ORDER BY project\_name)  
  
-- Index Recommendation:  
-- CREATE INDEX idx\_projects\_employee ON projects(employee\_id, project\_id, project\_name);  
  
-- =====================================================  
  
-- 3. Match orders with customers and display unmatched orders as well (Left Join)  
-- Approach: LEFT JOIN to show all orders, including orphaned ones  
-- Performance: Index on customer\_id for efficient join  
  
SELECT  
 o.order\_id,  
 o.order\_date,  
 o.order\_value,  
 o.customer\_id,  
 *COALESCE*(c.customer\_name, 'CUSTOMER NOT FOUND') as customer\_name,  
 c.region,  
 c.registration\_date,  
 -- Data quality indicators  
 CASE  
 WHEN c.customer\_id IS NULL THEN 'ORPHANED ORDER'  
 WHEN c.registration\_date > o.order\_date THEN 'DATA INCONSISTENCY'  
 ELSE 'VALID ORDER'  
 END as order\_status,  
 -- Customer metrics  
 CASE  
 WHEN c.customer\_id IS NOT NULL THEN  
 *DATEDIFF*(o.order\_date, c.registration\_date) -- Days since registration  
 ELSE NULL  
 END as days\_since\_registration  
FROM orders o  
LEFT JOIN customers c ON o.customer\_id = c.customer\_id  
ORDER BY  
 CASE WHEN c.customer\_id IS NULL THEN 0 ELSE 1 END, -- Orphaned orders first  
 o.order\_date DESC;  
  
-- Summary of unmatched orders:  
WITH order\_analysis AS (  
 SELECT  
 o.order\_id,  
 o.customer\_id,  
 o.order\_value,  
 c.customer\_name,  
 CASE WHEN c.customer\_id IS NULL THEN 1 ELSE 0 END as is\_orphaned  
 FROM orders o  
 LEFT JOIN customers c ON o.customer\_id = c.customer\_id  
)  
SELECT  
 *COUNT*(\*) as total\_orders,  
 *SUM*(is\_orphaned) as orphaned\_orders,  
 *ROUND*((*SUM*(is\_orphaned) \* 100.0 / *COUNT*(\*)), 2) as orphaned\_percentage,  
 *SUM*(CASE WHEN is\_orphaned = 1 THEN order\_value ELSE 0 END) as orphaned\_order\_value  
FROM order\_analysis;  
  
-- Index Recommendations:  
-- CREATE INDEX idx\_orders\_customer ON orders(customer\_id, order\_date, order\_value);  
-- CREATE INDEX idx\_customers\_id\_name ON customers(customer\_id, customer\_name);  
  
-- =====================================================  
  
-- 4. Create unique product combinations using a Cross Join while excluding identical product pairs  
-- Approach: CROSS JOIN with filtering to avoid duplicates and self-pairs  
-- Performance: Use WHERE clause efficiently to reduce result set  
  
SELECT  
 p1.product\_id as product1\_id,  
 p1.product\_name as product1\_name,  
 p1.category as product1\_category,  
 p1.price as product1\_price,  
 p2.product\_id as product2\_id,  
 p2.product\_name as product2\_name,  
 p2.category as product2\_category,  
 p2.price as product2\_price,  
 -- Combination metrics  
 *ABS*(p1.price - p2.price) as price\_difference,  
 (p1.price + p2.price) as bundle\_price,  
 CASE  
 WHEN p1.category = p2.category THEN 'Same Category'  
 ELSE 'Cross Category'  
 END as combination\_type,  
 -- Suggested bundle discount  
 *ROUND*((p1.price + p2.price) \* 0.9, 2) as discounted\_bundle\_price  
FROM products p1  
CROSS JOIN products p2  
WHERE p1.product\_id < p2.product\_id -- Ensure unique pairs and avoid self-pairs  
 AND p1.category != p2.category -- Optional: only cross-category combinations  
ORDER BY  
 combination\_type,  
 bundle\_price DESC;  
  
-- Alternative: Include same-category combinations but with different logic  
SELECT  
 p1.product\_id as product1\_id,  
 p1.product\_name as product1\_name,  
 p2.product\_id as product2\_id,  
 p2.product\_name as product2\_name,  
 p1.category as category,  
 (p1.price + p2.price) as bundle\_price,  
 -- Compatibility score based on price similarity  
 CASE  
 WHEN *ABS*(p1.price - p2.price) <= 10 THEN 'High Compatibility'  
 WHEN *ABS*(p1.price - p2.price) <= 50 THEN 'Medium Compatibility'  
 ELSE 'Low Compatibility'  
 END as price\_compatibility  
FROM products p1  
CROSS JOIN products p2  
WHERE p1.product\_id != p2.product\_id -- Exclude identical products  
 AND p1.product\_id < p2.product\_id -- Avoid duplicate pairs (A,B) and (B,A)  
ORDER BY bundle\_price DESC;  
  
-- Performance note: CROSS JOIN can be expensive with large tables  
-- Consider adding LIMIT clause for initial analysis  
-- Index Recommendation:  
-- CREATE INDEX idx\_products\_category\_price ON products(category, price, product\_id);  
  
-- =====================================================  
  
-- 5. Retrieve employees along with their direct managers using a self-join  
-- Approach: Self-join on employees table using manager\_id  
-- Performance: Index on manager\_id for efficient hierarchical queries  
  
SELECT  
 e.employee\_id,  
 e.first\_name as employee\_first\_name,  
 e.last\_name as employee\_last\_name,  
 e.salary as employee\_salary,  
 e.department\_id,  
 e.manager\_id,  
 m.employee\_id as manager\_employee\_id,  
 m.first\_name as manager\_first\_name,  
 m.last\_name as manager\_last\_name,  
 m.salary as manager\_salary,  
 -- Hierarchy analysis  
 CASE  
 WHEN e.manager\_id IS NULL THEN 'TOP LEVEL'  
 WHEN m.manager\_id IS NULL THEN 'REPORTS TO CEO'  
 ELSE 'MID LEVEL'  
 END as hierarchy\_level,  
 -- Salary comparison  
 CASE  
 WHEN e.manager\_id IS NULL THEN NULL  
 WHEN e.salary > m.salary THEN 'EMPLOYEE EARNS MORE'  
 WHEN e.salary = m.salary THEN 'EQUAL SALARY'  
 ELSE 'NORMAL HIERARCHY'  
 END as salary\_comparison,  
 -- Management span  
 *COALESCE*(m.direct\_reports, 0) as manager\_direct\_reports  
FROM employees e  
LEFT JOIN employees m ON e.manager\_id = m.employee\_id  
LEFT JOIN (  
 -- Count direct reports for each manager  
 SELECT  
 manager\_id,  
 *COUNT*(\*) as direct\_reports  
 FROM employees  
 WHERE manager\_id IS NOT NULL  
 GROUP BY manager\_id  
) dr ON m.employee\_id = dr.manager\_id  
ORDER BY  
 CASE WHEN e.manager\_id IS NULL THEN 0 ELSE 1 END, -- Top level first  
 m.last\_name,  
 e.last\_name;  
  
-- Hierarchical query with levels (recursive approach for deeper hierarchy):  
WITH RECURSIVE employee\_hierarchy AS (  
 -- Base case: Top-level employees (no manager)  
 SELECT  
 employee\_id,  
 first\_name,  
 last\_name,  
 manager\_id,  
 salary,  
 1 as level,  
 *CAST*(last\_name AS VARCHAR(1000)) as hierarchy\_path  
 FROM employees  
 WHERE manager\_id IS NULL  
  
 UNION ALL  
  
 -- Recursive case: Employees with managers  
 SELECT  
 e.employee\_id,  
 e.first\_name,  
 e.last\_name,  
 e.manager\_id,  
 e.salary,  
 eh.level + 1,  
 *CAST*(eh.hierarchy\_path || ' -> ' || e.last\_name AS VARCHAR(1000))  
 FROM employees e  
 JOIN employee\_hierarchy eh ON e.manager\_id = eh.employee\_id  
 WHERE eh.level < 10 -- Prevent infinite recursion  
)  
SELECT  
 employee\_id,  
 *REPEAT*(' ', level - 1) || first\_name || ' ' || last\_name as indented\_name,  
 level,  
 hierarchy\_path,  
 salary  
FROM employee\_hierarchy  
ORDER BY hierarchy\_path;  
  
-- Index Recommendations:  
-- CREATE INDEX idx\_employees\_manager ON employees(manager\_id, employee\_id);  
-- CREATE INDEX idx\_employees\_dept\_manager ON employees(department\_id, manager\_id);  
  
-- =====================================================  
-- SUBQUERIES (4 Questions)  
-- =====================================================  
  
-- 1. Find customers whose total purchase value exceeds the average order value  
-- Approach: Correlated subquery to compare customer total vs average  
-- Performance: Consider materialized view for frequently accessed aggregations  
  
SELECT  
 c.customer\_id,  
 c.customer\_name,  
 c.region,  
 customer\_totals.total\_purchase\_value,  
 customer\_totals.order\_count,  
 *ROUND*(customer\_totals.total\_purchase\_value / customer\_totals.order\_count, 2) as avg\_order\_value,  
 overall\_avg.company\_avg\_order\_value,  
 *ROUND*(  
 (customer\_totals.total\_purchase\_value / overall\_avg.company\_avg\_order\_value - 1) \* 100,  
 2  
 ) as percentage\_above\_average  
FROM customers c  
JOIN (  
 -- Customer totals subquery  
 SELECT  
 customer\_id,  
 *SUM*(order\_value) as total\_purchase\_value,  
 *COUNT*(\*) as order\_count  
 FROM orders  
 GROUP BY customer\_id  
) customer\_totals ON c.customer\_id = customer\_totals.customer\_id  
CROSS JOIN (  
 -- Overall average subquery  
 SELECT *AVG*(order\_value) as company\_avg\_order\_value  
 FROM orders  
) overall\_avg  
WHERE customer\_totals.total\_purchase\_value > overall\_avg.company\_avg\_order\_value  
ORDER BY customer\_totals.total\_purchase\_value DESC;  
  
-- Alternative with window function (single pass):  
WITH customer\_analysis AS (  
 SELECT  
 c.customer\_id,  
 c.customer\_name,  
 c.region,  
 *SUM*(o.order\_value) as total\_purchase\_value,  
 *COUNT*(o.order\_id) as order\_count,  
 *AVG*(*SUM*(o.order\_value)) OVER () as avg\_customer\_total,  
 *AVG*(o.order\_value) OVER () as avg\_order\_value  
 FROM customers c  
 JOIN orders o ON c.customer\_id = o.customer\_id  
 GROUP BY c.customer\_id, c.customer\_name, c.region  
)  
SELECT  
 customer\_id,  
 customer\_name,  
 region,  
 total\_purchase\_value,  
 order\_count,  
 avg\_order\_value,  
 *ROUND*((total\_purchase\_value / avg\_customer\_total - 1) \* 100, 2) as pct\_above\_avg\_customer  
FROM customer\_analysis  
WHERE total\_purchase\_value > avg\_customer\_total  
ORDER BY total\_purchase\_value DESC;  
  
-- Index Recommendations:  
-- CREATE INDEX idx\_orders\_customer\_value ON orders(customer\_id, order\_value);  
  
-- =====================================================  
  
-- 2. Retrieve employees with the lowest salary in their respective departments  
-- Approach: Correlated subquery to find minimum salary per department  
-- Performance: Index on (department\_id, salary) for efficient subquery execution  
  
SELECT  
 e.employee\_id,  
 e.first\_name,  
 e.last\_name,  
 e.department\_id,  
 e.salary,  
 e.hire\_date,  
 -- Additional context  
 dept\_stats.avg\_dept\_salary,  
 dept\_stats.max\_dept\_salary,  
 dept\_stats.employee\_count,  
 *ROUND*((dept\_stats.avg\_dept\_salary - e.salary), 2) as salary\_gap\_from\_avg  
FROM employees e  
JOIN (  
 -- Department statistics subquery  
 SELECT  
 department\_id,  
 *MIN*(salary) as min\_salary,  
 *AVG*(salary) as avg\_dept\_salary,  
 *MAX*(salary) as max\_dept\_salary,  
 *COUNT*(\*) as employee\_count  
 FROM employees  
 GROUP BY department\_id  
) dept\_stats ON e.department\_id = dept\_stats.department\_id  
 AND e.salary = dept\_stats.min\_salary  
ORDER BY e.department\_id, e.hire\_date;  
  
-- Alternative using window function:  
WITH salary\_ranks AS (  
 SELECT  
 employee\_id,  
 first\_name,  
 last\_name,  
 department\_id,  
 salary,  
 hire\_date,  
 *ROW\_NUMBER*() OVER (PARTITION BY department\_id ORDER BY salary ASC, hire\_date ASC) as salary\_rank,  
 *MIN*(salary) OVER (PARTITION BY department\_id) as dept\_min\_salary,  
 *AVG*(salary) OVER (PARTITION BY department\_id) as dept\_avg\_salary  
 FROM employees  
)  
SELECT  
 employee\_id,  
 first\_name,  
 last\_name,  
 department\_id,  
 salary,  
 hire\_date,  
 dept\_avg\_salary,  
 *ROUND*(dept\_avg\_salary - salary, 2) as salary\_gap\_from\_avg  
FROM salary\_ranks  
WHERE salary\_rank = 1 -- Lowest salary (with tie-breaking by hire\_date)  
ORDER BY department\_id;  
  
-- Multiple employees with same lowest salary:  
SELECT  
 e.employee\_id,  
 e.first\_name,  
 e.last\_name,  
 e.department\_id,  
 e.salary,  
 e.hire\_date  
FROM employees e  
WHERE e.salary = (  
 SELECT *MIN*(salary)  
 FROM employees e2  
 WHERE e2.department\_id = e.department\_id  
)  
ORDER BY e.department\_id, e.hire\_date;  
  
-- Index Recommendation:  
-- CREATE INDEX idx\_employees\_dept\_salary\_hire ON employees(department\_id, salary, hire\_date);  
  
-- =====================================================  
  
-- 3. Identify products ordered more than 10 times using a subquery  
-- Approach: EXISTS subquery with HAVING clause for efficient filtering  
-- Performance: Index on product\_id for fast aggregation  
  
SELECT  
 p.product\_id,  
 p.product\_name,  
 p.category,  
 p.price,  
 product\_stats.order\_count,  
 product\_stats.total\_quantity,  
 product\_stats.total\_revenue,  
 product\_stats.avg\_order\_value,  
 product\_stats.first\_order\_date,  
 product\_stats.last\_order\_date  
FROM products p  
JOIN (  
 -- Product order statistics subquery  
 SELECT  
 product\_id,  
 *COUNT*(\*) as order\_count,  
 *SUM*(quantity) as total\_quantity, -- Assuming quantity column exists  
 *SUM*(order\_value) as total\_revenue,  
 *AVG*(order\_value) as avg\_order\_value,  
 *MIN*(order\_date) as first\_order\_date,  
 *MAX*(order\_date) as last\_order\_date  
 FROM orders  
 GROUP BY product\_id  
 HAVING *COUNT*(\*) > 10  
) product\_stats ON p.product\_id = product\_stats.product\_id  
ORDER BY product\_stats.order\_count DESC;  
  
-- Alternative using EXISTS (more readable):  
SELECT  
 p.product\_id,  
 p.product\_name,  
 p.category,  
 p.price,  
 (SELECT *COUNT*(\*) FROM orders o WHERE o.product\_id = p.product\_id) as order\_count  
FROM products p  
WHERE *EXISTS* (  
 SELECT 1  
 FROM orders o  
 WHERE o.product\_id = p.product\_id  
 GROUP BY o.product\_id  
 HAVING *COUNT*(\*) > 10  
)  
ORDER BY order\_count DESC;  
  
-- Performance analysis query:  
WITH product\_performance AS (  
 SELECT  
 p.product\_id,  
 p.product\_name,  
 p.category,  
 *COUNT*(o.order\_id) as order\_count,  
 *SUM*(o.order\_value) as total\_revenue,  
 *COUNT*(DISTINCT o.customer\_id) as unique\_customers,  
 *AVG*(o.order\_value) as avg\_order\_value  
 FROM products p  
 LEFT JOIN orders o ON p.product\_id = o.product\_id  
 GROUP BY p.product\_id, p.product\_name, p.category  
)  
SELECT  
 product\_id,  
 product\_name,  
 category,  
 order\_count,  
 total\_revenue,  
 unique\_customers,  
 avg\_order\_value,  
 CASE  
 WHEN order\_count > 50 THEN 'High Demand'  
 WHEN order\_count > 10 THEN 'Medium Demand'  
 WHEN order\_count > 0 THEN 'Low Demand'  
 ELSE 'No Orders'  
 END as demand\_category  
FROM product\_performance  
WHERE order\_count > 10  
ORDER BY order\_count DESC;  
  
-- Index Recommendations:  
-- CREATE INDEX idx\_orders\_product\_date ON orders(product\_id, order\_date, order\_value);  
-- CREATE INDEX idx\_products\_category\_name ON products(category, product\_name);  
  
-- =====================================================  
  
-- 4. List regions where the highest sales value is below a specified threshold  
-- Approach: Subquery to find max sales per region, filter by threshold  
-- Performance: Index on (region, sale\_amount) for efficient MAX calculation  
  
-- Method 1: Using subquery with MAX  
SELECT  
 region\_data.region,  
 region\_data.max\_sale\_value,  
 region\_data.total\_sales,  
 region\_data.sale\_count,  
 region\_data.avg\_sale\_value,  
 region\_data.min\_sale\_value  
FROM (  
 SELECT  
 region,  
 *MAX*(sale\_amount) as max\_sale\_value,  
 *SUM*(sale\_amount) as total\_sales,  
 *COUNT*(\*) as sale\_count,  
 *AVG*(sale\_amount) as avg\_sale\_value,  
 *MIN*(sale\_amount) as min\_sale\_value  
 FROM sales  
 GROUP BY region  
) region\_data  
WHERE region\_data.max\_sale\_value < 5000 -- Threshold: $5,000  
ORDER BY region\_data.max\_sale\_value DESC;  
  
-- Method 2: Using window function for additional context  
WITH region\_analysis AS (  
 SELECT  
 region,  
 sale\_amount,  
 sale\_date,  
 *MAX*(sale\_amount) OVER (PARTITION BY region) as region\_max\_sale,  
 *AVG*(sale\_amount) OVER (PARTITION BY region) as region\_avg\_sale,  
 *COUNT*(\*) OVER (PARTITION BY region) as region\_sale\_count,  
 *ROW\_NUMBER*() OVER (PARTITION BY region ORDER BY sale\_amount DESC) as sale\_rank  
 FROM sales  
),  
region\_summary AS (  
 SELECT DISTINCT  
 region,  
 region\_max\_sale,  
 region\_avg\_sale,  
 region\_sale\_count,  
 -- Performance metrics  
 CASE  
 WHEN region\_max\_sale < 1000 THEN 'Very Low Performance'  
 WHEN region\_max\_sale < 5000 THEN 'Low Performance'  
 WHEN region\_max\_sale < 10000 THEN 'Medium Performance'  
 ELSE 'High Performance'  
 END as performance\_category  
 FROM region\_analysis  
)  
SELECT  
 region,  
 region\_max\_sale,  
 region\_avg\_sale,  
 region\_sale\_count,  
 performance\_category,  
 -- Recommendations  
 CASE  
 WHEN region\_max\_sale < 5000 THEN 'Needs attention: Consider training or market analysis'  
 ELSE 'Performing well'  
 END as recommendation  
FROM region\_summary  
WHERE region\_max\_sale < 5000 -- Configurable threshold  
ORDER BY region\_max\_sale ASC;  
  
-- Method 3: Parameterized version for stored procedure  
-- DECLARE @threshold DECIMAL(10,2) = 5000;  
--  
-- SELECT  
-- s.region,  
-- MAX(s.sale\_amount) as max\_sale\_value,  
-- COUNT(\*) as total\_sales,  
-- AVG(s.sale\_amount) as avg\_sale\_value  
-- FROM sales s  
-- GROUP BY s.region  
-- HAVING MAX(s.sale\_amount) < @threshold  
-- ORDER BY MAX(s.sale\_amount) DESC;  
  
-- Additional analysis: Compare with overall company performance  
WITH region\_performance AS (  
 SELECT  
 region,  
 *MAX*(sale\_amount) as max\_sale\_value,  
 *SUM*(sale\_amount) as total\_sales,  
 *COUNT*(\*) as sale\_count  
 FROM sales  
 GROUP BY region  
),  
company\_benchmark AS (  
 SELECT  
 *AVG*(max\_sale\_value) as company\_avg\_max\_sale,  
 *PERCENTILE\_CONT*(0.5) WITHIN GROUP (ORDER BY max\_sale\_value) as company\_median\_max\_sale  
 FROM region\_performance  
)  
SELECT  
 rp.region,  
 rp.max\_sale\_value,  
 rp.total\_sales,  
 rp.sale\_count,  
 cb.company\_avg\_max\_sale,  
 *ROUND*((rp.max\_sale\_value / cb.company\_avg\_max\_sale - 1) \* 100, 2) as pct\_vs\_company\_avg  
FROM region\_performance rp  
CROSS JOIN company\_benchmark cb  
WHERE rp.max\_sale\_value < 5000  
ORDER BY rp.max\_sale\_value DESC;  
  
-- Index Recommendations:  
-- CREATE INDEX idx\_sales\_region\_amount ON sales(region, sale\_amount DESC);  
-- CREATE INDEX idx\_sales\_region\_date\_amount ON sales(region, sale\_date, sale\_amount);  
  
-- =====================================================  
-- AGGREGATE FUNCTIONS (5 Questions)  
-- =====================================================  
  
-- 1. Compute the median salary for each department  
-- Approach: Use PERCENTILE\_CONT for accurate median calculation  
-- Cross-platform note: Median functions vary across databases  
  
-- PostgreSQL/SQL Server/Oracle Solution:  
SELECT  
 department\_id,  
 *COUNT*(\*) as employee\_count,  
 *MIN*(salary) as min\_salary,  
 *PERCENTILE\_CONT*(0.5) WITHIN GROUP (ORDER BY salary) as median\_salary,  
 AVG(salary) as mean\_salary,  
 MAX(salary) as max\_salary,  
 STDDEV(salary) as salary\_std\_dev,  
 -- Additional percentiles for salary distribution analysis  
 PERCENTILE\_CONT(0.25) WITHIN GROUP (ORDER BY salary) as q1\_salary,  
 PERCENTILE\_CONT(0.75) WITHIN GROUP (ORDER BY salary) as q3\_salary,  
 PERCENTILE\_CONT(0.9) WITHIN GROUP (ORDER BY salary) as p90\_salary  
FROM employees  
WHERE salary IS NOT NULL  
GROUP BY department\_id  
ORDER BY median\_salary DESC;  
  
-- MySQL Alternative (no PERCENTILE\_CONT):  
-- SELECT  
-- department\_id,  
-- COUNT(\*) as employee\_count,  
-- MIN(salary) as min\_salary,  
-- CASE  
-- WHEN COUNT(\*) % 2 = 1 THEN  
-- (SELECT salary FROM employees e2  
-- WHERE e2.department\_id = e1.department\_id  
-- ORDER BY salary  
-- LIMIT 1 OFFSET (COUNT(\*) DIV 2))  
-- ELSE  
-- (SELECT AVG(salary) FROM (  
-- SELECT salary FROM employees e3  
-- WHERE e3.department\_id = e1.department\_id  
-- ORDER BY salary  
-- LIMIT 2 OFFSET (COUNT(\*) DIV 2 - 1)  
-- ) t)  
-- END as median\_salary,  
-- AVG(salary) as mean\_salary,  
-- MAX(salary) as max\_salary  
-- FROM employees e1  
-- WHERE salary IS NOT NULL  
-- GROUP BY department\_id  
-- ORDER BY median\_salary DESC;  
  
-- Alternative using window functions (works on most modern databases):  
WITH salary\_rankings AS (  
 SELECT  
 department\_id,  
 salary,  
 *ROW\_NUMBER*() OVER (PARTITION BY department\_id ORDER BY salary) as row\_num,  
 *COUNT*(\*) OVER (PARTITION BY department\_id) as total\_count  
 FROM employees  
 WHERE salary IS NOT NULL  
),  
median\_calc AS (  
 SELECT  
 department\_id,  
 *AVG*(salary) as median\_salary  
 FROM salary\_rankings  
 WHERE row\_num IN (  
 (total\_count + 1) / 2, -- For odd count  
 (total\_count + 2) / 2 -- For even count  
 )  
 GROUP BY department\_id  
)  
SELECT  
 mc.department\_id,  
 *COUNT*(e.employee\_id) as employee\_count,  
 *MIN*(e.salary) as min\_salary,  
 mc.median\_salary,  
 *AVG*(e.salary) as mean\_salary,  
 *MAX*(e.salary) as max\_salary,  
 -- Salary distribution analysis  
 *ROUND*(mc.median\_salary - *AVG*(e.salary), 2) as median\_mean\_diff  
FROM median\_calc mc  
JOIN employees e ON mc.department\_id = e.department\_id  
WHERE e.salary IS NOT NULL  
GROUP BY mc.department\_id, mc.median\_salary  
ORDER BY mc.median\_salary DESC;  
  
-- Index Recommendation:  
-- CREATE INDEX idx\_employees\_dept\_salary ON employees(department\_id, salary);  
  
-- =====================================================  
  
-- 2. Summarize monthly sales and rank them in descending order  
-- Approach: Extract month/year, aggregate sales, apply ranking functions  
-- Performance: Index on sale\_date for efficient date-based grouping  
  
SELECT  
 sales\_year,  
 sales\_month,  
 month\_name,  
 total\_sales,  
 sale\_count,  
 avg\_sale\_amount,  
 unique\_employees,  
 -- Ranking metrics  
 *RANK*() OVER (ORDER BY total\_sales DESC) as sales\_rank,  
 *DENSE\_RANK*() OVER (ORDER BY total\_sales DESC) as dense\_sales\_rank,  
 *ROW\_NUMBER*() OVER (ORDER BY total\_sales DESC) as unique\_rank,  
 -- Performance analysis  
 *ROUND*((total\_sales / *SUM*(total\_sales) OVER ()) \* 100, 2) as market\_share\_pct,  
 *LAG*(total\_sales) OVER (ORDER BY sales\_year, sales\_month) as prev\_month\_sales,  
 *ROUND*(  
 ((total\_sales - *LAG*(total\_sales) OVER (ORDER BY sales\_year, sales\_month)) /  
 *LAG*(total\_sales) OVER (ORDER BY sales\_year, sales\_month)) \* 100,  
 2  
 ) as month\_over\_month\_growth  
FROM (  
 SELECT  
 *EXTRACT*(YEAR FROM sale\_date) as sales\_year,  
 *EXTRACT*(MONTH FROM sale\_date) as sales\_month,  
 *TO\_CHAR*(sale\_date, 'Month') as month\_name, -- PostgreSQL  
 -- DATENAME(MONTH, sale\_date) as month\_name, -- SQL Server  
 -- TO\_CHAR(sale\_date, 'Month') as month\_name, -- Oracle  
 *SUM*(sale\_amount) as total\_sales,  
 *COUNT*(\*) as sale\_count,  
 *AVG*(sale\_amount) as avg\_sale\_amount,  
 *COUNT*(DISTINCT employee\_id) as unique\_employees  
 FROM sales  
 GROUP BY  
 *EXTRACT*(YEAR FROM sale\_date),  
 *EXTRACT*(MONTH FROM sale\_date),  
 *TO\_CHAR*(sale\_date, 'Month')  
) monthly\_summary  
ORDER BY sales\_rank;  
  
-- Cross-platform date extraction:  
-- PostgreSQL: EXTRACT(YEAR FROM date), DATE\_TRUNC('month', date)  
-- MySQL: YEAR(date), MONTH(date), DATE\_FORMAT(date, '%Y-%m')  
-- SQL Server: YEAR(date), MONTH(date), FORMAT(date, 'yyyy-MM')  
-- Oracle: EXTRACT(YEAR FROM date), TRUNC(date, 'MM')  
  
-- Alternative with moving averages:  
WITH monthly\_sales AS (  
 SELECT  
 *DATE\_TRUNC*('month', sale\_date) as sale\_month,  
 *SUM*(sale\_amount) as monthly\_total,  
 *COUNT*(\*) as transaction\_count,  
 *COUNT*(DISTINCT employee\_id) as active\_employees  
 FROM sales  
 GROUP BY *DATE\_TRUNC*('month', sale\_date)  
),  
sales\_with\_trends AS (  
 SELECT  
 sale\_month,  
 monthly\_total,  
 transaction\_count,  
 active\_employees,  
 -- Trend analysis  
 *AVG*(monthly\_total) OVER (  
 ORDER BY sale\_month  
 ROWS BETWEEN 2 PRECEDING AND CURRENT ROW  
 ) as three\_month\_avg,  
 *AVG*(monthly\_total) OVER (  
 ORDER BY sale\_month  
 ROWS BETWEEN 11 PRECEDING AND CURRENT ROW  
 ) as twelve\_month\_avg,  
 -- Ranking  
 *RANK*() OVER (ORDER BY monthly\_total DESC) as rank\_by\_total  
 FROM monthly\_sales  
)  
SELECT  
 *TO\_CHAR*(sale\_month, 'YYYY-MM') as month\_year,  
 monthly\_total,  
 transaction\_count,  
 active\_employees,  
 rank\_by\_total,  
 *ROUND*(three\_month\_avg, 2) as three\_month\_avg,  
 *ROUND*(twelve\_month\_avg, 2) as twelve\_month\_avg,  
 -- Performance indicators  
 CASE  
 WHEN monthly\_total > twelve\_month\_avg \* 1.1 THEN 'Above Average'  
 WHEN monthly\_total < twelve\_month\_avg \* 0.9 THEN 'Below Average'  
 ELSE 'Average'  
 END as performance\_category  
FROM sales\_with\_trends  
ORDER BY rank\_by\_total;  
  
-- Index Recommendations:  
-- CREATE INDEX idx\_sales\_date\_amount\_emp ON sales(sale\_date, sale\_amount, employee\_id);  
  
-- =====================================================  
  
-- 3. Count the number of unique customers for each product  
-- Approach: COUNT(DISTINCT) with comprehensive product analysis  
-- Performance: Index on (product\_id, customer\_id) for efficient distinct counting  
  
SELECT  
 p.product\_id,  
 p.product\_name,  
 p.category,  
 p.price,  
 *COUNT*(DISTINCT o.customer\_id) as unique\_customers,  
 *COUNT*(o.order\_id) as total\_orders,  
 *SUM*(o.order\_value) as total\_revenue,  
 *AVG*(o.order\_value) as avg\_order\_value,  
 *MIN*(o.order\_date) as first\_order\_date,  
 *MAX*(o.order\_date) as last\_order\_date,  
 -- Customer penetration metrics  
 *ROUND*(  
 *COUNT*(DISTINCT o.customer\_id) \* 100.0 /  
 (SELECT *COUNT*(DISTINCT customer\_id) FROM orders),  
 2  
 ) as customer\_penetration\_pct,  
 -- Repeat customer analysis  
 *ROUND*(  
 *COUNT*(o.order\_id) \* 1.0 / *COUNT*(DISTINCT o.customer\_id),  
 2  
 ) as avg\_orders\_per\_customer  
FROM products p  
LEFT JOIN orders o ON p.product\_id = o.product\_id  
GROUP BY p.product\_id, p.product\_name, p.category, p.price  
ORDER BY unique\_customers DESC, total\_revenue DESC;  
  
-- Advanced analysis with customer segmentation:  
WITH product\_customer\_analysis AS (  
 SELECT  
 p.product\_id,  
 p.product\_name,  
 p.category,  
 *COUNT*(DISTINCT o.customer\_id) as unique\_customers,  
 *COUNT*(o.order\_id) as total\_orders,  
 *SUM*(o.order\_value) as total\_revenue,  
 -- Customer value segmentation  
 *COUNT*(DISTINCT CASE WHEN customer\_totals.customer\_value > 1000 THEN o.customer\_id END) as high\_value\_customers,  
 *COUNT*(DISTINCT CASE WHEN customer\_totals.customer\_value BETWEEN 500 AND 1000 THEN o.customer\_id END) as medium\_value\_customers,  
 *COUNT*(DISTINCT CASE WHEN customer\_totals.customer\_value < 500 THEN o.customer\_id END) as low\_value\_customers  
 FROM products p  
 LEFT JOIN orders o ON p.product\_id = o.product\_id  
 LEFT JOIN (  
 SELECT  
 customer\_id,  
 *SUM*(order\_value) as customer\_value  
 FROM orders  
 GROUP BY customer\_id  
 ) customer\_totals ON o.customer\_id = customer\_totals.customer\_id  
 GROUP BY p.product\_id, p.product\_name, p.category  
)  
SELECT  
 product\_id,  
 product\_name,  
 category,  
 unique\_customers,  
 total\_orders,  
 total\_revenue,  
 high\_value\_customers,  
 medium\_value\_customers,  
 low\_value\_customers,  
 -- Customer mix analysis  
 *ROUND*(high\_value\_customers \* 100.0 / *NULLIF*(unique\_customers, 0), 2) as high\_value\_customer\_pct,  
 -- Product popularity ranking  
 *RANK*() OVER (ORDER BY unique\_customers DESC) as popularity\_rank,  
 *RANK*() OVER (ORDER BY total\_revenue DESC) as revenue\_rank  
FROM product\_customer\_analysis  
WHERE unique\_customers > 0  
ORDER BY unique\_customers DESC;  
  
-- Performance optimization for large datasets:  
-- Consider using approximate count distinct for very large tables  
-- PostgreSQL: SELECT APPROX\_COUNT\_DISTINCT(customer\_id)  
-- SQL Server: SELECT APPROX\_COUNT\_DISTINCT(customer\_id)  
  
-- Index Recommendations:  
-- CREATE INDEX idx\_orders\_product\_customer ON orders(product\_id, customer\_id, order\_value);  
-- CREATE INDEX idx\_products\_category\_id ON products(category, product\_id);  
  
-- =====================================================  
  
-- 4. Identify the top five regions based on total sales  
-- Approach: Aggregate by region, rank and limit to top 5  
-- Performance: Index on (region, sale\_amount) for efficient aggregation  
  
WITH region\_performance AS (  
 SELECT  
 region,  
 *SUM*(sale\_amount) as total\_sales,  
 *COUNT*(\*) as sale\_count,  
 *COUNT*(DISTINCT employee\_id) as active\_employees,  
 *AVG*(sale\_amount) as avg\_sale\_amount,  
 *MIN*(sale\_amount) as min\_sale\_amount,  
 *MAX*(sale\_amount) as max\_sale\_amount,  
 *STDDEV*(sale\_amount) as sale\_amount\_stddev,  
 *MIN*(sale\_date) as first\_sale\_date,  
 *MAX*(sale\_date) as last\_sale\_date  
 FROM sales  
 GROUP BY region  
),  
region\_rankings AS (  
 SELECT  
 \*,  
 *RANK*() OVER (ORDER BY total\_sales DESC) as sales\_rank,  
 *RANK*() OVER (ORDER BY avg\_sale\_amount DESC) as avg\_sale\_rank,  
 *RANK*() OVER (ORDER BY sale\_count DESC) as volume\_rank,  
 -- Market share calculation  
 *ROUND*((total\_sales / *SUM*(total\_sales) OVER ()) \* 100, 2) as market\_share\_pct,  
 -- Performance consistency (coefficient of variation)  
 *ROUND*((sale\_amount\_stddev / avg\_sale\_amount) \* 100, 2) as coefficient\_of\_variation  
 FROM region\_performance  
)  
SELECT  
 sales\_rank,  
 region,  
 total\_sales,  
 market\_share\_pct,  
 sale\_count,  
 active\_employees,  
 avg\_sale\_amount,  
 avg\_sale\_rank,  
 volume\_rank,  
 coefficient\_of\_variation,  
 first\_sale\_date,  
 last\_sale\_date,  
 -- Performance indicators  
 CASE  
 WHEN coefficient\_of\_variation < 50 THEN 'Consistent'  
 WHEN coefficient\_of\_variation < 100 THEN 'Moderate Variance'  
 ELSE 'High Variance'  
 END as performance\_consistency,  
 -- Sales productivity  
 *ROUND*(total\_sales / active\_employees, 2) as sales\_per\_employee  
FROM region\_rankings  
WHERE sales\_rank <= 5  
ORDER BY sales\_rank;  
  
-- Alternative with additional metrics:  
SELECT  
 region,  
 total\_sales,  
 sale\_count,  
 avg\_sale\_amount,  
 active\_employees,  
 -- Growth analysis (requires historical data)  
 *LAG*(total\_sales) OVER (ORDER BY region) as prev\_period\_sales,  
 -- Efficiency metrics  
 *ROUND*(total\_sales / sale\_count, 2) as revenue\_per\_transaction,  
 *ROUND*(total\_sales / active\_employees, 2) as revenue\_per\_employee,  
 -- Regional performance score (weighted)  
 *ROUND*(  
 (total\_sales \* 0.4 + avg\_sale\_amount \* sale\_count \* 0.3 + active\_employees \* avg\_sale\_amount \* 0.3) /  
 (SELECT *MAX*(total\_sales) FROM (  
 SELECT *SUM*(sale\_amount) as total\_sales  
 FROM sales  
 GROUP BY region  
 ) max\_sales) \* 100,  
 2  
 ) as performance\_score  
FROM (  
 SELECT  
 region,  
 *SUM*(sale\_amount) as total\_sales,  
 *COUNT*(\*) as sale\_count,  
 *AVG*(sale\_amount) as avg\_sale\_amount,  
 *COUNT*(DISTINCT employee\_id) as active\_employees  
 FROM sales  
 GROUP BY region  
) region\_summary  
ORDER BY total\_sales DESC  
LIMIT 5;  
  
-- Comparative analysis with benchmarks:  
WITH region\_stats AS (  
 SELECT  
 region,  
 *SUM*(sale\_amount) as total\_sales,  
 *COUNT*(\*) as sale\_count,  
 *AVG*(sale\_amount) as avg\_sale\_amount  
 FROM sales  
 GROUP BY region  
),  
benchmarks AS (  
 SELECT  
 *AVG*(total\_sales) as avg\_region\_sales,  
 *PERCENTILE\_CONT*(0.8) WITHIN GROUP (ORDER BY total\_sales) as top\_20\_threshold  
 FROM region\_stats  
)  
SELECT  
 rs.region,  
 rs.total\_sales,  
 rs.sale\_count,  
 rs.avg\_sale\_amount,  
 b.avg\_region\_sales,  
 *ROUND*((rs.total\_sales / b.avg\_region\_sales - 1) \* 100, 2) as vs\_average\_pct,  
 CASE  
 WHEN rs.total\_sales >= b.top\_20\_threshold THEN 'Top Performer'  
 WHEN rs.total\_sales >= b.avg\_region\_sales THEN 'Above Average'  
 ELSE 'Below Average'  
 END as performance\_category,  
 *ROW\_NUMBER*() OVER (ORDER BY rs.total\_sales DESC) as rank  
FROM region\_stats rs  
CROSS JOIN benchmarks b  
WHERE *ROW\_NUMBER*() OVER (ORDER BY rs.total\_sales DESC) <= 5  
ORDER BY rs.total\_sales DESC;  
  
-- Index Recommendations:  
-- CREATE INDEX idx\_sales\_region\_amount\_date ON sales(region, sale\_amount, sale\_date);  
-- CREATE INDEX idx\_sales\_region\_employee ON sales(region, employee\_id, sale\_amount);  
  
-- =====================================================  
  
-- 5. Calculate the average order value for every customer  
-- Approach: GROUP BY customer with comprehensive customer analysis  
-- Performance: Index on (customer\_id, order\_value) for efficient grouping  
  
SELECT  
 c.customer\_id,  
 c.customer\_name,  
 c.region,  
 c.registration\_date,  
 *COALESCE*(customer\_metrics.order\_count, 0) as total\_orders,  
 *COALESCE*(customer\_metrics.total\_order\_value, 0) as total\_spent,  
 *COALESCE*(customer\_metrics.avg\_order\_value, 0) as avg\_order\_value,  
 customer\_metrics.min\_order\_value,  
 customer\_metrics.max\_order\_value,  
 customer\_metrics.first\_order\_date,  
 customer\_metrics.last\_order\_date,  
 -- Customer lifecycle analysis  
 CASE  
 WHEN customer\_metrics.order\_count IS NULL THEN 'No Orders'  
 WHEN customer\_metrics.order\_count = 1 THEN 'Single Purchase'  
 WHEN customer\_metrics.order\_count <= 5 THEN 'Occasional'  
 WHEN customer\_metrics.order\_count <= 15 THEN 'Regular'  
 ELSE 'Frequent'  
 END as customer\_segment,  
 -- Time-based metrics  
 CASE  
 WHEN customer\_metrics.first\_order\_date IS NOT NULL THEN  
 *EXTRACT*(DAY FROM (customer\_metrics.last\_order\_date - customer\_metrics.first\_order\_date))  
 ELSE NULL  
 END as customer\_lifespan\_days,  
 -- Value-based ranking  
 *RANK*() OVER (ORDER BY customer\_metrics.avg\_order\_value DESC) as avg\_order\_value\_rank,  
 *RANK*() OVER (ORDER BY customer\_metrics.total\_order\_value DESC) as total\_value\_rank  
FROM customers c  
LEFT JOIN (  
 SELECT  
 customer\_id,  
 *COUNT*(\*) as order\_count,  
 *SUM*(order\_value) as total\_order\_value,  
 *AVG*(order\_value) as avg\_order\_value,  
 *MIN*(order\_value) as min\_order\_value,  
 *MAX*(order\_value) as max\_order\_value,  
 *MIN*(order\_date) as first\_order\_date,  
 *MAX*(order\_date) as last\_order\_date,  
 *STDDEV*(order\_value) as order\_value\_stddev  
 FROM orders  
 GROUP BY customer\_id  
) customer\_metrics ON c.customer\_id = customer\_metrics.customer\_id  
ORDER BY customer\_metrics.avg\_order\_value DESC NULLS LAST;  
  
-- Advanced customer analysis with percentiles:  
WITH customer\_order\_analysis AS (  
 SELECT  
 c.customer\_id,  
 c.customer\_name,  
 c.region,  
 c.registration\_date,  
 *COUNT*(o.order\_id) as order\_count,  
 *SUM*(o.order\_value) as total\_order\_value,  
 *AVG*(o.order\_value) as avg\_order\_value,  
 -- Percentile analysis of order values  
 *PERCENTILE\_CONT*(0.5) WITHIN GROUP (ORDER BY o.order\_value) as median\_order\_value,  
 PERCENTILE\_CONT(0.25) WITHIN GROUP (ORDER BY o.order\_value) as q1\_order\_value,  
 PERCENTILE\_CONT(0.75) WITHIN GROUP (ORDER BY o.order\_value) as q3\_order\_value,  
 MIN(o.order\_value) as min\_order\_value,  
 MAX(o.order\_value) as max\_order\_value  
 FROM customers c  
 LEFT JOIN orders o ON c.customer\_id = o.customer\_id  
 GROUP BY c.customer\_id, c.customer\_name, c.region, c.registration\_date  
),  
customer\_benchmarks AS (  
 SELECT  
 AVG(avg\_order\_value) as company\_avg\_order\_value,  
 PERCENTILE\_CONT(0.8) WITHIN GROUP (ORDER BY avg\_order\_value) as top\_20\_avg\_threshold  
 FROM customer\_order\_analysis  
 WHERE order\_count > 0  
)  
SELECT  
 coa.customer\_id,  
 coa.customer\_name,  
 coa.region,  
 coa.order\_count,  
 coa.total\_order\_value,  
 coa.avg\_order\_value,  
 coa.median\_order\_value,  
 cb.company\_avg\_order\_value,  
 -- Performance vs company average  
 *ROUND*(  
 (coa.avg\_order\_value / cb.company\_avg\_order\_value - 1) \* 100,  
 2  
 ) as vs\_company\_avg\_pct,  
 -- Customer value tier  
 CASE  
 WHEN coa.avg\_order\_value >= cb.top\_20\_avg\_threshold THEN 'Premium'  
 WHEN coa.avg\_order\_value >= cb.company\_avg\_order\_value THEN 'Standard'  
 WHEN coa.order\_count > 0 THEN 'Economy'  
 ELSE 'Inactive'  
 END as customer\_tier,  
 -- Order consistency (coefficient of variation)  
 CASE  
 WHEN coa.order\_count > 1 THEN  
 *ROUND*(((coa.q3\_order\_value - coa.q1\_order\_value) / coa.median\_order\_value) \* 100, 2)  
 ELSE NULL  
 END as order\_consistency\_score  
FROM customer\_order\_analysis coa  
CROSS JOIN customer\_benchmarks cb  
ORDER BY coa.avg\_order\_value DESC NULLS LAST;  
  
-- Customer lifetime value prediction:  
WITH customer\_metrics AS (  
 SELECT  
 customer\_id,  
 *COUNT*(\*) as order\_count,  
 *AVG*(order\_value) as avg\_order\_value,  
 *SUM*(order\_value) as total\_value,  
 *MAX*(order\_date) as last\_order\_date,  
 *MIN*(order\_date) as first\_order\_date,  
 -- Order frequency (orders per month)  
 CASE  
 WHEN *MAX*(order\_date) != MIN(order\_date) THEN  
 COUNT(\*) \* 30.0 / EXTRACT(DAY FROM (MAX(order\_date) - MIN(order\_date)))  
 ELSE NULL  
 END as orders\_per\_month  
 FROM orders  
 GROUP BY customer\_id  
)  
SELECT  
 c.customer\_id,  
 c.customer\_name,  
 c.region,  
 cm.order\_count,  
 cm.avg\_order\_value,  
 cm.total\_value,  
 cm.orders\_per\_month,  
 -- Predicted annual value (simplified)  
 CASE  
 WHEN cm.orders\_per\_month IS NOT NULL THEN  
 *ROUND*(cm.orders\_per\_month \* 12 \* cm.avg\_order\_value, 2)  
 ELSE NULL  
 END as predicted\_annual\_value,  
 -- Recency analysis  
 *EXTRACT*(DAY FROM (*CURRENT\_DATE* - cm.last\_order\_date)) as days\_since\_last\_order,  
 CASE  
 WHEN *EXTRACT*(DAY FROM (*CURRENT\_DATE* - cm.last\_order\_date)) <= 30 THEN 'Active'  
 WHEN *EXTRACT*(DAY FROM (*CURRENT\_DATE* - cm.last\_order\_date)) <= 90 THEN 'At Risk'  
 ELSE 'Inactive'  
 END as customer\_status  
FROM customers c  
JOIN customer\_metrics cm ON c.customer\_id = cm.customer\_id  
ORDER BY cm.avg\_order\_value DESC;  
  
-- Index Recommendations:  
-- CREATE INDEX idx\_orders\_customer\_value\_date ON orders(customer\_id, order\_value, order\_date);  
-- CREATE INDEX idx\_customers\_region\_reg\_date ON customers(region, registration\_date);  
  
-- =====================================================  
-- INDEXING AND PERFORMANCE (5 Questions)  
-- =====================================================  
  
-- 1. Write a query to locate duplicate entries in a column with an index  
-- Approach: Leverage index for efficient duplicate detection  
-- Performance: Demonstrate index usage vs full table scan  
  
-- Create index for demonstration (commented out - would be created separately):  
-- CREATE INDEX idx\_customers\_email ON customers(email);  
-- CREATE INDEX idx\_customers\_phone ON customers(phone);  
  
-- Efficient duplicate detection using indexed columns:  
SELECT  
 email,  
 *COUNT*(\*) as duplicate\_count,  
 *STRING\_AGG*(customer\_id::TEXT, ', ' ORDER BY customer\_id) as customer\_ids,  
 *MIN*(customer\_id) as keep\_customer\_id,  
 *MAX*(customer\_id) as latest\_customer\_id  
FROM customers  
WHERE email IS NOT NULL  
GROUP BY email  
HAVING *COUNT*(\*) > 1  
ORDER BY duplicate\_count DESC, email;  
  
-- Performance comparison query (with execution plan analysis):  
-- Method 1: Using indexed column (efficient)  
EXPLAIN (ANALYZE, BUFFERS)  
SELECT email, *COUNT*(\*)  
FROM customers  
WHERE email IS NOT NULL  
GROUP BY email  
HAVING *COUNT*(\*) > 1;  
  
-- Method 2: Using non-indexed column (slower)  
-- EXPLAIN (ANALYZE, BUFFERS)  
-- SELECT customer\_name, COUNT(\*)  
-- FROM customers  
-- WHERE customer\_name IS NOT NULL  
-- GROUP BY customer\_name  
-- HAVING COUNT(\*) > 1;  
  
-- Advanced duplicate analysis with index utilization:  
WITH duplicate\_analysis AS (  
 SELECT  
 email,  
 customer\_id,  
 customer\_name,  
 registration\_date,  
 *ROW\_NUMBER*() OVER (PARTITION BY email ORDER BY registration\_date ASC, customer\_id ASC) as rn,  
 *COUNT*(\*) OVER (PARTITION BY email) as duplicate\_count  
 FROM customers  
 WHERE email IS NOT NULL  
)  
SELECT  
 email,  
 duplicate\_count,  
 customer\_id,  
 customer\_name,  
 registration\_date,  
 CASE  
 WHEN rn = 1 THEN 'KEEP - Original'  
 ELSE 'DUPLICATE - Consider removal'  
 END as action\_recommendation,  
 -- Data quality metrics  
 CASE  
 WHEN duplicate\_count > 5 THEN 'High duplication'  
 WHEN duplicate\_count > 2 THEN 'Medium duplication'  
 ELSE 'Low duplication'  
 END as duplication\_severity  
FROM duplicate\_analysis  
WHERE duplicate\_count > 1  
ORDER BY duplicate\_count DESC, email, rn;  
  
-- Index effectiveness query:  
-- This query shows how the index is being used  
SELECT  
 schemaname,  
 tablename,  
 indexname,  
 idx\_scan as index\_scans,  
 idx\_tup\_read as tuples\_read,  
 idx\_tup\_fetch as tuples\_fetched  
FROM pg\_stat\_user\_indexes  
WHERE tablename = 'customers'  
 AND indexname LIKE '%email%';  
  
-- Alternative for SQL Server:  
-- SELECT  
-- i.name as index\_name,  
-- s.user\_seeks,  
-- s.user\_scans,  
-- s.user\_lookups,  
-- s.user\_updates  
-- FROM sys.dm\_db\_index\_usage\_stats s  
-- JOIN sys.indexes i ON s.object\_id = i.object\_id AND s.index\_id = i.index\_id  
-- WHERE OBJECT\_NAME(s.object\_id) = 'customers';  
  
-- =====================================================  
  
-- 2. Evaluate the effect of a composite index on query performance  
-- Approach: Compare queries with and without composite index usage  
-- Performance: Demonstrate covering index benefits  
  
-- Composite index creation (commented - would be created separately):  
-- CREATE INDEX idx\_orders\_customer\_date\_value ON orders(customer\_id, order\_date, order\_value);  
-- CREATE INDEX idx\_orders\_covering ON orders(customer\_id, order\_date) INCLUDE (order\_value, product\_id);  
  
-- Query optimized for composite index usage:  
EXPLAIN (ANALYZE, BUFFERS)  
SELECT  
 customer\_id,  
 *COUNT*(\*) as order\_count,  
 *SUM*(order\_value) as total\_value,  
 *AVG*(order\_value) as avg\_value,  
 *MIN*(order\_date) as first\_order,  
 *MAX*(order\_date) as last\_order  
FROM orders  
WHERE customer\_id BETWEEN 1000 AND 2000  
 AND order\_date >= '2023-01-01'  
 AND order\_date < '2024-01-01'  
GROUP BY customer\_id  
ORDER BY customer\_id;  
  
-- Performance comparison: Range query vs equality  
-- This query benefits from the composite index leading column  
EXPLAIN (ANALYZE, BUFFERS)  
SELECT \*  
FROM orders  
WHERE customer\_id = 1500  
 AND order\_date BETWEEN '2023-06-01' AND '2023-06-30'  
ORDER BY order\_date;  
  
-- Index coverage analysis:  
WITH index\_usage AS (  
 SELECT  
 customer\_id,  
 order\_date,  
 order\_value,  
 -- This query should use index-only scan with covering index  
 *ROW\_NUMBER*() OVER (PARTITION BY customer\_id ORDER BY order\_date DESC) as recent\_order\_rank  
 FROM orders  
 WHERE customer\_id IN (1001, 1002, 1003, 1004, 1005)  
)  
SELECT  
 customer\_id,  
 order\_date,  
 order\_value  
FROM index\_usage  
WHERE recent\_order\_rank <= 3  
ORDER BY customer\_id, recent\_order\_rank;  
  
-- Composite index effectiveness test:  
-- Test different WHERE clause orders to show index column order importance  
-- Query 1: Follows index column order (efficient)  
EXPLAIN (ANALYZE, BUFFERS)  
SELECT *COUNT*(\*)  
FROM orders  
WHERE customer\_id > 1000  
 AND order\_date > '2023-01-01'  
 AND order\_value > 100;  
  
-- Query 2: Doesn't follow index column order (less efficient)  
EXPLAIN (ANALYZE, BUFFERS)  
SELECT *COUNT*(\*)  
FROM orders  
WHERE order\_date > '2023-01-01'  
 AND order\_value > 100  
 AND customer\_id > 1000;  
  
-- Index statistics query to show composite index usage:  
SELECT  
 schemaname,  
 tablename,  
 indexname,  
 idx\_scan,  
 idx\_tup\_read,  
 idx\_tup\_fetch,  
 -- Index efficiency ratio  
 CASE  
 WHEN idx\_tup\_read > 0 THEN  
 *ROUND*((idx\_tup\_fetch::DECIMAL / idx\_tup\_read) \* 100, 2)  
 ELSE 0  
 END as efficiency\_ratio  
FROM pg\_stat\_user\_indexes  
WHERE tablename = 'orders'  
ORDER BY idx\_scan DESC;  
  
-- Performance benchmark query:  
-- Shows the difference between using and not using the composite index  
WITH performance\_test AS (  
 SELECT  
 customer\_id,  
 *DATE\_TRUNC*('month', order\_date) as order\_month,  
 *SUM*(order\_value) as monthly\_total,  
 *COUNT*(\*) as monthly\_orders  
 FROM orders  
 WHERE customer\_id BETWEEN 1000 AND 5000 -- Uses index efficiently  
 AND order\_date >= '2023-01-01'  
 AND order\_date < '2024-01-01'  
 GROUP BY customer\_id, *DATE\_TRUNC*('month', order\_date)  
)  
SELECT  
 customer\_id,  
 order\_month,  
 monthly\_total,  
 monthly\_orders,  
 *AVG*(monthly\_total) OVER (  
 PARTITION BY customer\_id  
 ORDER BY order\_month  
 ROWS BETWEEN 2 PRECEDING AND CURRENT ROW  
 ) as rolling\_avg  
FROM performance\_test  
ORDER BY customer\_id, order\_month;  
  
-- =====================================================  
  
-- 3. Identify high-cardinality columns that could benefit from indexing  
-- Approach: Analyze column statistics to identify indexing candidates  
-- Performance: Query system catalogs for cardinality analysis  
  
-- PostgreSQL column cardinality analysis:  
WITH column\_stats AS (  
 SELECT  
 schemaname,  
 tablename,  
 attname as column\_name,  
 n\_distinct,  
 correlation,  
 -- Estimate actual cardinality  
 CASE  
 WHEN n\_distinct > 0 THEN n\_distinct  
 WHEN n\_distinct < 0 THEN *ABS*(n\_distinct) \* reltuples  
 ELSE NULL  
 END as estimated\_distinct\_values,  
 reltuples as table\_rows  
 FROM pg\_stats ps  
 JOIN pg\_class pc ON ps.tablename = pc.relname  
 WHERE schemaname = 'public'  
 AND tablename IN ('customers', 'orders', 'employees', 'products', 'sales')  
),  
index\_candidates AS (  
 SELECT  
 tablename,  
 column\_name,  
 estimated\_distinct\_values,  
 table\_rows,  
 -- Cardinality ratio (selectivity)  
 CASE  
 WHEN table\_rows > 0 THEN  
 *ROUND*((estimated\_distinct\_values / table\_rows) \* 100, 2)  
 ELSE 0  
 END as cardinality\_ratio,  
 correlation,  
 -- Index recommendation score  
 CASE  
 WHEN estimated\_distinct\_values / table\_rows > 0.1 THEN 'High Priority'  
 WHEN estimated\_distinct\_values / table\_rows > 0.05 THEN 'Medium Priority'  
 WHEN estimated\_distinct\_values / table\_rows > 0.01 THEN 'Low Priority'  
 ELSE 'Not Recommended'  
 END as index\_recommendation  
 FROM column\_stats  
 WHERE estimated\_distinct\_values IS NOT NULL  
 AND table\_rows > 1000 -- Only consider tables with sufficient data  
)  
SELECT  
 tablename,  
 column\_name,  
 estimated\_distinct\_values,  
 table\_rows,  
 cardinality\_ratio,  
 correlation,  
 index\_recommendation,  
 -- Specific recommendations  
 CASE  
 WHEN cardinality\_ratio > 50 THEN 'Excellent candidate for B-tree index'  
 WHEN cardinality\_ratio > 10 THEN 'Good candidate for B-tree index'  
 WHEN cardinality\_ratio > 1 AND correlation > 0.1 THEN 'Consider for composite index'  
 WHEN cardinality\_ratio < 1 THEN 'Consider bitmap index (if supported)'  
 ELSE 'Low cardinality - not suitable for indexing'  
 END as detailed\_recommendation  
FROM index\_candidates  
ORDER BY cardinality\_ratio DESC, table\_rows DESC;  
  
-- Cross-platform alternative using actual queries:  
-- This approach works on all database systems  
WITH table\_analysis AS (  
 -- Analyze customers table  
 SELECT  
 'customers' as table\_name,  
 'customer\_id' as column\_name,  
 *COUNT*(DISTINCT customer\_id) as distinct\_values,  
 *COUNT*(\*) as total\_rows,  
 'Primary Key' as column\_type  
 FROM customers  
  
 UNION ALL  
  
 SELECT  
 'customers',  
 'region',  
 *COUNT*(DISTINCT region),  
 *COUNT*(\*),  
 'Categorical'  
 FROM customers  
  
 UNION ALL  
  
 SELECT  
 'customers',  
 'customer\_name',  
 *COUNT*(DISTINCT customer\_name),  
 *COUNT*(\*),  
 'Text'  
 FROM customers  
  
 UNION ALL  
  
 -- Analyze orders table  
 SELECT  
 'orders',  
 'customer\_id',  
 *COUNT*(DISTINCT customer\_id),  
 *COUNT*(\*),  
 'Foreign Key'  
 FROM orders  
  
 UNION ALL  
  
 SELECT  
 'orders',  
 'order\_date',  
 *COUNT*(DISTINCT order\_date),  
 *COUNT*(\*),  
 'Date'  
 FROM orders  
  
 UNION ALL  
  
 SELECT  
 'orders',  
 'order\_value',  
 *COUNT*(DISTINCT order\_value),  
 *COUNT*(\*),  
 'Numeric'  
 FROM orders  
)  
SELECT  
 table\_name,  
 column\_name,  
 column\_type,  
 distinct\_values,  
 total\_rows,  
 *ROUND*((distinct\_values::DECIMAL / total\_rows) \* 100, 2) as selectivity\_pct,  
 CASE  
 WHEN distinct\_values::DECIMAL / total\_rows > 0.8 THEN 'Unique - Primary/Unique index'  
 WHEN distinct\_values::DECIMAL / total\_rows > 0.1 THEN 'High selectivity - Good for indexing'  
 WHEN distinct\_values::DECIMAL / total\_rows > 0.01 THEN 'Medium selectivity - Consider composite index'  
 ELSE 'Low selectivity - Not suitable for B-tree index'  
 END as index\_suitability,  
 -- Estimated index size (rough calculation)  
 CASE  
 WHEN column\_type = 'Text' THEN total\_rows \* 20 -- Assume avg 20 bytes  
 WHEN column\_type = 'Date' THEN total\_rows \* 8  
 WHEN column\_type = 'Numeric' THEN total\_rows \* 8  
 ELSE total\_rows \* 4  
 END as estimated\_index\_size\_bytes  
FROM table\_analysis  
ORDER BY selectivity\_pct DESC;  
  
-- Query to identify columns used in WHERE clauses (for index recommendations):  
-- This would typically be done by analyzing query logs  
SELECT  
 'orders' as table\_name,  
 'Frequently filtered columns' as analysis\_type,  
 'customer\_id, order\_date, order\_value' as recommended\_indexes,  
 'CREATE INDEX idx\_orders\_customer\_date ON orders(customer\_id, order\_date);  
 CREATE INDEX idx\_orders\_date\_value ON orders(order\_date, order\_value);' as suggested\_ddl  
  
UNION ALL  
  
SELECT  
 'customers',  
 'Lookup columns',  
 'email, phone, region',  
 'CREATE INDEX idx\_customers\_email ON customers(email);  
 CREATE INDEX idx\_customers\_region ON customers(region);'  
  
UNION ALL  
  
SELECT  
 'employees',  
 'Hierarchy and salary analysis',  
 'department\_id, manager\_id, salary',  
 'CREATE INDEX idx\_employees\_dept\_salary ON employees(department\_id, salary);  
 CREATE INDEX idx\_employees\_manager ON employees(manager\_id);';  
  
-- =====================================================  
  
-- 4. Compare query execution times before and after implementing a clustered index  
-- Approach: Demonstrate clustered index impact on range queries and sorting  
-- Performance: Show physical data organization benefits  
  
-- Clustered index creation (commented - implementation specific):  
-- SQL Server: CREATE CLUSTERED INDEX idx\_orders\_date\_clustered ON orders(order\_date);  
-- PostgreSQL: CLUSTER orders USING idx\_orders\_date; (after creating B-tree index)  
  
-- Baseline query performance (before clustered index):  
-- Range query that benefits from clustered index  
EXPLAIN (ANALYZE, BUFFERS)  
SELECT  
 order\_id,  
 customer\_id,  
 order\_date,  
 order\_value  
FROM orders  
WHERE order\_date BETWEEN '2023-06-01' AND '2023-06-30'  
ORDER BY order\_date;  
  
-- Performance test query - date range aggregation:  
EXPLAIN (ANALYZE, BUFFERS)  
WITH monthly\_aggregates AS (  
 SELECT  
 DATE\_TRUNC('day', order\_date) as order\_day,  
 COUNT(\*) as daily\_orders,  
 SUM(order\_value) as daily\_revenue,  
 AVG(order\_value) as avg\_order\_value  
 FROM orders  
 WHERE order\_date >= '2023-01-01'  
 AND order\_date < '2024-01-01'  
 GROUP BY DATE\_TRUNC('day', order\_date)  
)  
SELECT  
 order\_day,  
 daily\_orders,  
 daily\_revenue,  
 avg\_order\_value,  
 *SUM*(daily\_revenue) OVER (ORDER BY order\_day) as cumulative\_revenue  
FROM monthly\_aggregates  
ORDER BY order\_day;  
  
-- Sequential scan performance test:  
-- This query should show significant improvement with clustered index  
EXPLAIN (ANALYZE, BUFFERS)  
SELECT  
 *COUNT*(\*) as total\_orders,  
 *SUM*(order\_value) as total\_revenue,  
 *MIN*(order\_date) as first\_date,  
 *MAX*(order\_date) as last\_date  
FROM orders  
WHERE order\_date >= '2023-01-01'  
 AND order\_date <= '2023-12-31';  
  
-- Page access pattern analysis (PostgreSQL specific):  
-- Shows physical I/O improvement with clustered data  
SELECT  
 heap\_blks\_read,  
 heap\_blks\_hit,  
 idx\_blks\_read,  
 idx\_blks\_hit,  
 -- Cache hit ratio  
 *ROUND*(  
 (heap\_blks\_hit::DECIMAL / (heap\_blks\_hit + heap\_blks\_read)) \* 100,  
 2  
 ) as heap\_hit\_ratio,  
 *ROUND*(  
 (idx\_blks\_hit::DECIMAL / (idx\_blks\_hit + idx\_blks\_read)) \* 100,  
 2  
 ) as index\_hit\_ratio  
FROM pg\_statio\_user\_tables  
WHERE relname = 'orders';  
  
-- Clustering effectiveness query:  
-- Measures how well the physical order matches the logical order  
SELECT  
 schemaname,  
 tablename,  
 indexname,  
 correlation  
FROM pg\_stats  
WHERE tablename = 'orders'  
 AND attname = 'order\_date';  
  
-- Performance comparison framework:  
-- Create a timing function to measure query performance  
DO $$  
DECLARE  
 start\_time TIMESTAMP;  
 end\_time TIMESTAMP;  
 execution\_time INTERVAL;  
BEGIN  
 -- Time the range query  
 start\_time := clock\_timestamp();  
  
 PERFORM COUNT(\*)  
 FROM orders  
 WHERE order\_date BETWEEN '2023-06-01' AND '2023-06-30';  
  
 end\_time := clock\_timestamp();  
 execution\_time := end\_time - start\_time;  
  
 RAISE NOTICE 'Range query execution time: %', execution\_time;  
END $$;  
  
-- Alternative SQL Server performance comparison:  
-- SET STATISTICS TIME ON;  
-- SET STATISTICS IO ON;  
--  
-- SELECT COUNT(\*)  
-- FROM orders  
-- WHERE order\_date BETWEEN '2023-06-01' AND '2023-06-30';  
--  
-- SET STATISTICS TIME OFF;  
-- SET STATISTICS IO OFF;  
  
-- Clustered index maintenance analysis:  
SELECT  
 schemaname,  
 tablename,  
 indexname,  
 idx\_scan,  
 idx\_tup\_read,  
 idx\_tup\_fetch,  
 -- Index maintenance cost indicator  
 CASE  
 WHEN idx\_scan > 0 THEN  
 *ROUND*(idx\_tup\_read::DECIMAL / idx\_scan, 2)  
 ELSE 0  
 END as avg\_tuples\_per\_scan  
FROM pg\_stat\_user\_indexes  
WHERE tablename = 'orders'  
 AND indexname LIKE '%clustered%'  
ORDER BY idx\_scan DESC;  
  
-- =====================================================  
  
-- 5. Write a query that bypasses indexing to observe performance variations  
-- Approach: Force full table scan to demonstrate index value  
-- Performance: Use query hints and functions to disable index usage  
  
-- Force full table scan using functions that disable index usage:  
-- Method 1: Using functions on indexed columns  
EXPLAIN (ANALYZE, BUFFERS)  
SELECT  
 customer\_id,  
 customer\_name,  
 email  
FROM customers  
WHERE *UPPER*(email) = *UPPER*('john.doe@email.com') -- Function disables index on email  
 OR *LENGTH*(customer\_name) > 10; -- Function disables index on customer\_name  
  
-- Method 2: Using arithmetic operations on indexed columns:  
EXPLAIN (ANALYZE, BUFFERS)  
SELECT  
 order\_id,  
 customer\_id,  
 order\_date,  
 order\_value  
FROM orders  
WHERE customer\_id + 0 = 1500 -- Adding 0 disables index on customer\_id  
 AND order\_date + INTERVAL '0 days' BETWEEN '2023-01-01' AND '2023-12-31';  
  
-- Method 3: Using OR conditions that force full scan:  
EXPLAIN (ANALYZE, BUFFERS)  
SELECT  
 employee\_id,  
 first\_name,  
 last\_name,  
 salary  
FROM employees  
WHERE salary > 50000  
 OR employee\_id IS NOT NULL; -- This OR condition forces full scan  
  
-- Method 4: Using inequality operations on low-cardinality columns:  
EXPLAIN (ANALYZE, BUFFERS)  
SELECT  
 product\_id,  
 product\_name,  
 category,  
 price  
FROM products  
WHERE category != 'Electronics' -- Inequality on low-cardinality column  
 AND price != 0;  
  
-- Performance comparison: Index vs No Index usage  
-- Query 1: Optimized for index usage  
EXPLAIN (ANALYZE, BUFFERS)  
SELECT  
 customer\_id,  
 order\_date,  
 order\_value  
FROM orders  
WHERE customer\_id = 1500  
 AND order\_date BETWEEN '2023-06-01' AND '2023-06-30'  
ORDER BY order\_date;  
  
-- Query 2: Same logic but bypassing indexes  
EXPLAIN (ANALYZE, BUFFERS)  
SELECT  
 customer\_id,  
 order\_date,  
 order\_value  
FROM orders  
WHERE customer\_id \* 1 = 1500 -- Disable customer\_id index  
 AND *EXTRACT*(YEAR FROM order\_date) = 2023 -- Disable date index  
 AND *EXTRACT*(MONTH FROM order\_date) = 6  
ORDER BY order\_date + INTERVAL '0 seconds'; -- Disable ORDER BY optimization  
  
-- Using query hints to disable index usage (database specific):  
-- PostgreSQL: Use pg\_hint\_plan extension  
-- /\*+ SeqScan(orders) \*/  
-- SELECT \* FROM orders WHERE customer\_id = 1500;  
  
-- SQL Server example:  
-- SELECT \* FROM orders WITH (INDEX(0)) -- Force table scan  
-- WHERE customer\_id = 1500;  
  
-- Oracle example:  
-- SELECT /\*+ FULL(orders) \*/ \* FROM orders  
-- WHERE customer\_id = 1500;  
  
-- Performance monitoring query to show the difference:  
WITH index\_usage\_stats AS (  
 SELECT  
 schemaname,  
 tablename,  
 indexname,  
 idx\_scan as scans\_with\_index,  
 seq\_scan as sequential\_scans,  
 seq\_tup\_read as tuples\_read\_sequentially,  
 idx\_tup\_read as tuples\_read\_via\_index  
 FROM pg\_stat\_user\_indexes psi  
 JOIN pg\_stat\_user\_tables pst USING (schemaname, tablename)  
 WHERE tablename = 'orders'  
)  
SELECT  
 tablename,  
 indexname,  
 scans\_with\_index,  
 sequential\_scans,  
 -- Efficiency metrics  
 CASE  
 WHEN (scans\_with\_index + sequential\_scans) > 0 THEN  
 *ROUND*(  
 (scans\_with\_index::DECIMAL / (scans\_with\_index + sequential\_scans)) \* 100,  
 2  
 )  
 ELSE 0  
 END as index\_usage\_percentage,  
 tuples\_read\_via\_index,  
 tuples\_read\_sequentially,  
 -- Performance indicator  
 CASE  
 WHEN tuples\_read\_via\_index > 0 AND tuples\_read\_sequentially > 0 THEN  
 *ROUND*(tuples\_read\_sequentially::DECIMAL / tuples\_read\_via\_index, 2)  
 ELSE NULL  
 END as seq\_vs\_index\_ratio  
FROM index\_usage\_stats  
ORDER BY index\_usage\_percentage DESC;  
  
-- Demonstration of index selectivity impact:  
-- Low selectivity query (doesn't benefit much from index):  
EXPLAIN (ANALYZE, BUFFERS)  
SELECT *COUNT*(\*)  
FROM employees  
WHERE salary > 30000; -- If most employees earn > 30k, index won't help much  
  
-- High selectivity query (benefits significantly from index):  
EXPLAIN (ANALYZE, BUFFERS)  
SELECT \*  
FROM employees  
WHERE employee\_id = 12345; -- Unique lookup, index provides huge benefit  
  
-- Cost comparison query:  
-- Shows the query planner's cost estimates  
EXPLAIN (COSTS, BUFFERS)  
SELECT  
 o.order\_id,  
 o.order\_date,  
 c.customer\_name  
FROM orders o  
JOIN customers c ON o.customer\_id = c.customer\_id  
WHERE o.order\_date >= '2023-01-01'  
ORDER BY o.order\_date;  
  
-- Final performance summary:  
-- This query summarizes the impact of different query patterns  
SELECT  
 'Index-Optimized Query' as query\_type,  
 'Uses indexes effectively' as description,  
 'Low cost, fast execution' as expected\_performance  
  
UNION ALL  
  
SELECT  
 'Index-Bypassing Query',  
 'Forces full table scans',  
 'High cost, slow execution'  
  
UNION ALL  
  
SELECT  
 'Mixed Query',  
 'Some parts use indexes, others dont',  
 'Variable performance based on data distribution'  
  
UNION ALL  
  
SELECT  
 'Recommendation',  
 'Always test with representative data volumes',  
 'Monitor execution plans in production';  
  
-- =====================================================  
-- END OF SQL SOLUTIONS  
-- =====================================================  
  
-- PERFORMANCE OPTIMIZATION NOTES:  
-- 1. Always analyze execution plans before and after index creation  
-- 2. Consider data distribution and query patterns when designing indexes  
-- 3. Monitor index usage statistics to identify unused indexes  
-- 4. Use covering indexes for frequently accessed columns  
-- 5. Partition large tables by date or other logical boundaries  
-- 6. Update table statistics regularly for optimal query planning  
-- 7. Consider columnstore indexes for analytical workloads  
-- 8. Use appropriate data types to minimize storage and improve performance  
-- 9. Implement proper constraint checking for data integrity  
-- 10. Regular maintenance: REINDEX, UPDATE STATISTICS, ANALYZE TABLE  
  
-- CROSS-PLATFORM COMPATIBILITY NOTES:  
-- - PostgreSQL: Advanced features like LATERAL joins, array functions  
-- - MySQL: Different syntax for string functions, no FULL OUTER JOIN  
-- - SQL Server: T-SQL specific functions, excellent window function support  
-- - Oracle: Advanced analytical functions, hierarchical queries with CONNECT BY  
-- - SQLite: Limited window function support in older versions  
-- - Always test queries on target database platform  
-- - Use ANSI SQL standards when possible for maximum portability