SQL SKILLS GUIDE FOR DATA ANALYSTS

A Comprehensive Guide with Progressive Examples from Beginner to Advanced Level

---

TABLE OF CONTENTS

---

1. SELECT STATEMENTS & FILTERING

PURPOSE: SELECT is the foundation of all SQL queries. It allows you to retrieve specific data from tables, filter records based on conditions, and format output. Mastering SELECT and WHERE clauses is essential for data exploration and analysis.

Tables Used:

- employees (emp_id, name, department, salary, hire_date, manager_id)
- sales (sale_id, emp_id, sale_date, amount, region)

EASY: Basic Column Selection

Problem: Retrieve the names and departments of all employees.

```
SELECT name, department
FROM employees;
```

INTERMEDIATE: Multiple Conditions & Operators

Problem: Find all employees in the Sales or Marketing departments who earn more than $60,000 and were hired after January 1, 2020.

```
SELECT name, department, salary, hire_date
FROM employees
WHERE department IN ('Sales', 'Marketing')
  AND salary > 60000
  AND hire_date > '2020-01-01'
ORDER BY salary DESC;
```

ADVANCED: Complex Filtering with Pattern Matching

Problem: Find employees whose names start with 'A' or end with 'son', work in departments containing 'Tech', earn between $50K-$100K, and have been with the company for more than 2 years.

```
SELECT
    name,
    department,
    salary,
    hire_date,
    CURRENT_DATE - hire_date AS days_employed
FROM employees
WHERE (name LIKE 'A%' OR name LIKE '%son')
  AND department LIKE '%Tech%'
  AND salary BETWEEN 50000 AND 100000
  AND hire_date < CURRENT_DATE - INTERVAL '2 years'
ORDER BY days_employed DESC;
```

---

2. JOINS

PURPOSE: JOINs combine data from multiple tables based on related columns. This is critical in analytics because data is typically normalized across many tables. Understanding which type of JOIN to use determines what data you'll include or exclude from your analysis.

Tables Used:

- customers (customer_id, name, email, signup_date)
- orders (order_id, customer_id, order_date, total_amount)

- order_items (item_id, order_id, product_id, quantity)

EASY: Basic INNER JOIN

Problem: Show all customers who have placed orders along with their order totals.

```
SELECT
    c.name,
    c.email,
    o.order_id,
    o.total_amount
FROM customers c
INNER JOIN orders o ON c.customer_id = o.customer_id;
```

INTERMEDIATE: LEFT JOIN with Aggregation

Problem: List all customers including those who haven't ordered, showing their total number of orders and lifetime value. Include customers with 0 orders.

```
SELECT
    c.customer_id,
    c.name,
    COUNT(o.order_id) AS total_orders,
    COALESCE(SUM(o.total_amount), 0) AS lifetime_value
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.customer_id, c.name
ORDER BY lifetime_value DESC;
```

ADVANCED: Multiple JOINs with Conditions

Problem: Find customers who signed up in 2024, have placed more than 3 orders, with detailed item breakdown, but only for orders over $100. Show customer name, order count, and total items purchased.

```
SELECT
    c.customer_id,
    c.name,
    c.signup_date,
    COUNT(DISTINCT o.order_id) AS order_count,
    SUM(oi.quantity) AS total_items_purchased,
    SUM(o.total_amount) AS total_spent
FROM customers c
```

```
INNER JOIN orders o
    ON c.customer_id = o.customer_id
    AND o.total_amount > 100
INNER JOIN order_items oi
    ON o.order_id = oi.order_id
WHERE EXTRACT(YEAR FROM c.signup_date) = 2024
GROUP BY c.customer_id, c.name, c.signup_date
HAVING COUNT(DISTINCT o.order_id) > 3
ORDER BY total_spent DESC;
```

---

3. AGGREGATIONS & GROUP BY

PURPOSE: Aggregations summarize data by calculating totals, averages, counts, and other statistics. GROUP BY segments data into categories for analysis. This is fundamental for answering business questions like "What are our sales by region?" or "What's the average customer order value?"

Tables Used:

- products (product_id, name, category, price)
- sales (sale_id, product_id, sale_date, quantity, revenue)

EASY: Basic Aggregations

Problem: Calculate total revenue, average sale amount, and number of sales.

```
SELECT
    COUNT(*) AS total_sales,
    SUM(revenue) AS total_revenue,
    AVG(revenue) AS avg_sale_amount,
    MIN(revenue) AS smallest_sale,
    MAX(revenue) AS largest_sale
FROM sales;
```

INTERMEDIATE: GROUP BY with HAVING

Problem: Find product categories with more than $10,000 in total revenue and an average sale price above $50. Show category, total revenue, and number of products.

```
SELECT
    p.category,
```

```
    COUNT(DISTINCT p.product_id) AS product_count,
    SUM(s.revenue) AS total_revenue,
    AVG(s.revenue / s.quantity) AS avg_price_per_unit
FROM products p
INNER JOIN sales s ON p.product_id = s.product_id
GROUP BY p.category
HAVING SUM(s.revenue) > 10000
   AND AVG(s.revenue / s.quantity) > 50
ORDER BY total_revenue DESC;
```

ADVANCED: Complex Aggregations with Multiple Dimensions

Problem: Create a monthly sales report by category showing total revenue, units sold, average order value, and month-over-month growth percentage. Only include months with at least 10 sales.

```
WITH monthly_category_sales AS (
    SELECT
        DATE_TRUNC('month', s.sale_date) AS month,
        p.category,
        SUM(s.revenue) AS revenue,
        SUM(s.quantity) AS units_sold,
        COUNT(*) AS sale_count,
        AVG(s.revenue) AS avg_order_value
    FROM sales s
    INNER JOIN products p ON s.product_id = p.product_id
    GROUP BY DATE_TRUNC('month', s.sale_date), p.category
    HAVING COUNT(*) >= 10
)
SELECT
    month,
    category,
    revenue,
    units_sold,
    sale_count,
    ROUND(avg_order_value, 2) AS avg_order_value,
    LAG(revenue) OVER (PARTITION BY category ORDER BY month) AS
prev_month_revenue,
    ROUND(
        ((revenue - LAG(revenue) OVER (PARTITION BY category ORDER BY month)) /
         NULLIF(LAG(revenue) OVER (PARTITION BY category ORDER BY month), 0) * 100),
        2
    ) AS mom_growth_pct
```

```
FROM monthly_category_sales
ORDER BY month DESC, revenue DESC;
```

---

4. BASIC FUNCTIONS (STRING, DATE, CASE)

PURPOSE: Built-in functions manipulate and transform data. String functions clean text data, date functions enable time-based analysis, and CASE statements create conditional logic for categorization and business rules.

Tables Used:

- customers (customer_id, first_name, last_name, email, registration_date)
- transactions (trans_id, customer_id, trans_date, amount)

EASY: Basic String & Date Functions

Problem: Create a full name column, extract the email domain, and show how many days since registration.

```
SELECT
    customer_id,
    CONCAT(first_name, ' ', last_name) AS full_name,
    UPPER(email) AS email_upper,
    SUBSTRING(email FROM POSITION('@' IN email) + 1) AS email_domain,
    CURRENT_DATE - registration_date AS days_since_registration
FROM customers;
```

INTERMEDIATE: CASE Statements for Categorization

Problem: Categorize customers as 'New' (registered <30 days), 'Active' (30-365 days), or 'Veteran' (>365 days). Also categorize their spending as 'High', 'Medium', or 'Low' based on total transaction amounts.

```
SELECT
    c.customer_id,
    CONCAT(c.first_name, ' ', c.last_name) AS full_name,
    c.registration_date,
    CASE
        WHEN CURRENT_DATE - c.registration_date < 30 THEN 'New'
        WHEN CURRENT_DATE - c.registration_date <= 365 THEN 'Active'
        ELSE 'Veteran'
```

```
    END AS customer_segment,
    SUM(t.amount) AS total_spent,
    CASE
        WHEN SUM(t.amount) > 1000 THEN 'High Spender'
        WHEN SUM(t.amount) > 500 THEN 'Medium Spender'
        ELSE 'Low Spender'
    END AS spending_category
FROM customers c
LEFT JOIN transactions t ON c.customer_id = t.customer_id
GROUP BY c.customer_id, c.first_name, c.last_name, c.registration_date;
```

ADVANCED: Complex Date Analysis with Business Logic

Problem: Identify at-risk customers: those who registered more than 6 months ago, haven't transacted in 60+ days, but were previously active (3+ transactions in their first 90 days). Calculate their early activity rate vs recent inactivity period.

```
WITH customer_activity AS (
    SELECT
        c.customer_id,
        CONCAT(c.first_name, ' ', c.last_name) AS full_name,
        c.registration_date,
        MAX(t.trans_date) AS last_transaction,
        COUNT(CASE
            WHEN t.trans_date <= c.registration_date + INTERVAL '90 days'
            THEN 1
        END) AS early_trans_count,
        COUNT(t.trans_id) AS total_transactions,
        SUM(t.amount) AS lifetime_value
    FROM customers c
    LEFT JOIN transactions t ON c.customer_id = t.customer_id
    WHERE c.registration_date < CURRENT_DATE - INTERVAL '6 months'
    GROUP BY c.customer_id, c.first_name, c.last_name, c.registration_date
)
SELECT
    customer_id,
    full_name,
    registration_date,
    last_transaction,
    CURRENT_DATE - last_transaction AS days_since_last_trans,
    early_trans_count,
    total_transactions,
    ROUND(lifetime_value, 2) AS lifetime_value,
```

```
    CASE
        WHEN last_transaction IS NULL THEN 'Never Transacted'
        WHEN CURRENT_DATE - last_transaction > 60 AND early_trans_count >= 3
            THEN 'At Risk - Previously Active'
        WHEN CURRENT_DATE - last_transaction > 90
            THEN 'Churned'
        ELSE 'Active'
    END AS risk_status,
    ROUND(
        early_trans_count::NUMERIC /
        NULLIF(EXTRACT(DAY FROM registration_date + INTERVAL '90 days' -
registration_date), 0),
        3
    ) AS early_activity_rate
FROM customer_activity
WHERE early_trans_count >= 3
  AND (last_transaction IS NULL OR CURRENT_DATE - last_transaction > 60)
ORDER BY lifetime_value DESC;
```

---

5. SUBQUERIES (DEEP DIVE)

PURPOSE: Subqueries are queries nested within other queries. They're essential for comparing data against aggregates, filtering based on complex conditions, and breaking down complex problems into manageable steps.

KEY TYPES:

- Scalar Subqueries: Return a single value
- Column Subqueries: Return single column, multiple rows (used with IN, ANY, ALL)
- Table Subqueries: Return multiple rows and columns (used in FROM clause)
- Correlated Subqueries: Reference columns from outer query

Tables Used:

- employees (emp_id, name, department, salary, hire_date, manager_id)
- sales (sale_id, emp_id, product_id, sale_date, amount)
- products (product_id, name, category, cost)

EASY: Scalar Subquery in WHERE

Problem: Find all employees who earn more than the company average salary.

```
SELECT
    name,
    department,
    salary,
    salary - (SELECT AVG(salary) FROM employees) AS diff_from_avg
FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees)
ORDER BY salary DESC;
```

Explanation: The subquery `(SELECT AVG(salary) FROM employees)` calculates once and returns a single number. The outer query then compares each employee's salary against this value.

EASY: Subquery with IN Operator

Problem: List all employees who have made at least one sale.

```
SELECT
    emp_id,
    name,
    department
FROM employees
WHERE emp_id IN (
    SELECT DISTINCT emp_id
    FROM sales
);
```

Explanation: The subquery returns all unique employee IDs from the sales table. The IN operator checks if each employee's ID exists in this list.

INTERMEDIATE: Correlated Subquery

Problem: For each employee, show their rank within their department based on salary. Also show the department average.

```
SELECT
    e1.name,
    e1.department,
    e1.salary,
    (SELECT AVG(salary)
     FROM employees e2
     WHERE e2.department = e1.department) AS dept_avg_salary,
    (SELECT COUNT(*) + 1
```

```
    FROM employees e2
    WHERE e2.department = e1.department
      AND e2.salary > e1.salary) AS salary_rank_in_dept
FROM employees e1
ORDER BY e1.department, e1.salary DESC;
```

Explanation: A correlated subquery executes once for each row in the outer query. Notice how `e2.department = e1.department` connects the inner query to the outer query's current row.

INTERMEDIATE: Subquery in FROM Clause (Derived Table)

Problem: Find departments where the average salary is above $70,000, and show how many employees are in those departments earning above their department average.

```
SELECT
   dept_stats.department,
   dept_stats.avg_salary,
   dept_stats.emp_count,
   COUNT(e.emp_id) AS above_avg_count
FROM (
   SELECT
      department,
      AVG(salary) AS avg_salary,
      COUNT(*) AS emp_count
   FROM employees
   GROUP BY department
   HAVING AVG(salary) > 70000
) AS dept_stats
INNER JOIN employees e
   ON dept_stats.department = e.department
   AND e.salary > dept_stats.avg_salary
GROUP BY dept_stats.department, dept_stats.avg_salary, dept_stats.emp_count
ORDER BY dept_stats.avg_salary DESC;
```

Explanation: The subquery in the FROM clause creates a temporary table (dept_stats) that we can join to and query like any other table.

INTERMEDIATE: EXISTS vs IN

Problem: Find employees who have never made a sale (demonstrating NOT EXISTS).

```
-- Using NOT EXISTS (generally more efficient)
```

```
SELECT
    e.emp_id,
    e.name,
    e.department,
    e.hire_date
FROM employees e
WHERE NOT EXISTS (
    SELECT 1
    FROM sales s
    WHERE s.emp_id = e.emp_id
)
ORDER BY e.hire_date;
```

Explanation: EXISTS returns TRUE/FALSE and stops searching once it finds a match. The `SELECT 1` is a convention because EXISTS only cares whether rows exist, not what's in them.

ADVANCED: Multiple Nested Subqueries

Problem: Find products that have above-average sales in their category, but only in categories where the total category sales exceed $50,000.

```
SELECT
    p.product_id,
    p.name AS product_name,
    p.category,
    product_sales.total_sales,
    product_sales.sale_count,
    category_stats.category_avg_sales,
    category_stats.category_total_sales,
    ROUND(
        (product_sales.total_sales - category_stats.category_avg_sales) /
        category_stats.category_avg_sales * 100,
        2
    ) AS pct_above_category_avg
FROM products p
INNER JOIN (
    SELECT
        product_id,
        SUM(amount) AS total_sales,
        COUNT(*) AS sale_count
    FROM sales
    GROUP BY product_id
) AS product_sales ON p.product_id = product_sales.product_id
```

```
INNER JOIN (
    SELECT
        p2.category,
        AVG(product_totals.total_sales) AS category_avg_sales,
        SUM(product_totals.total_sales) AS category_total_sales
    FROM products p2
    INNER JOIN (
        SELECT
            product_id,
            SUM(amount) AS total_sales
        FROM sales
        GROUP BY product_id
    ) AS product_totals ON p2.product_id = product_totals.product_id
    GROUP BY p2.category
    HAVING SUM(product_totals.total_sales) > 50000
) AS category_stats ON p.category = category_stats.category
WHERE product_sales.total_sales > category_stats.category_avg_sales
ORDER BY category_stats.category, product_sales.total_sales DESC;
```

Explanation: This query has three layers of subqueries. Each layer builds on the previous: (1) Calculate product totals, (2) Calculate category statistics, (3) Join everything together.

ADVANCED: Subquery with ALL/ANY Operators

Problem: Find employees whose salary is higher than ALL employees in the 'Support' department.

```
-- Using ALL
SELECT
    name,
    department,
    salary,
    'Higher than ALL Support staff' AS comparison
FROM employees
WHERE salary > ALL (
    SELECT salary
    FROM employees
    WHERE department = 'Support'
);
```

Explanation: ALL means the condition must be true for every value returned by the subquery. ANY means the condition must be true for at least one value.

---

6. COMMON TABLE EXPRESSIONS - CTEs (DEEP DIVE)

PURPOSE: CTEs (WITH clauses) create named temporary result sets that exist only for the duration of a query. They make complex queries more readable, allow you to reference the same subquery multiple times, and enable recursive queries.

KEY ADVANTAGES:

- Readability: Break complex queries into logical, named steps
- Reusability: Reference the same CTE multiple times
- Maintainability: Easier to debug than nested subqueries
- Recursion: Enable recursive queries for hierarchical data

Tables Used:

- customers (customer_id, name, signup_date, segment)
- orders (order_id, customer_id, order_date, total_amount, status)
- order_items (item_id, order_id, product_id, quantity, price)
- products (product_id, name, category, cost)

EASY: Single CTE for Clarity

Problem: Calculate each customer's total spending and order count, then filter to show only customers who've spent more than $500.

```
WITH customer_spending AS (
    SELECT
        c.customer_id,
        c.name,
        SUM(o.total_amount) AS total_spent,
        COUNT(o.order_id) AS order_count
    FROM customers c
    INNER JOIN orders o ON c.customer_id = o.customer_id
    WHERE o.status = 'completed'
    GROUP BY c.customer_id, c.name
)
SELECT
    customer_id,
    name,
    total_spent,
    order_count
FROM customer_spending
```

```
WHERE total_spent > 500
ORDER BY total_spent DESC;
```

Explanation: The CTE separates data aggregation from filtering logic, making the query easier to understand and modify.

INTERMEDIATE: Multiple CTEs for Step-by-Step Logic

Problem: Build a customer segmentation analysis: identify "VIP" customers (top 20% by spending), calculate their average order frequency, and compare it to non-VIP customers.

```
WITH customer_metrics AS (
    SELECT
        c.customer_id,
        c.name,
        c.signup_date,
        COUNT(o.order_id) AS order_count,
        SUM(o.total_amount) AS lifetime_value,
        MIN(o.order_date) AS first_order,
        MAX(o.order_date) AS last_order
    FROM customers c
    LEFT JOIN orders o ON c.customer_id = o.customer_id
    WHERE o.status = 'completed' OR o.status IS NULL
    GROUP BY c.customer_id, c.name, c.signup_date
),
vip_threshold AS (
    SELECT PERCENTILE_CONT(0.8) WITHIN GROUP (ORDER BY lifetime_value) AS
threshold
    FROM customer_metrics
    WHERE lifetime_value > 0
),
segmented_customers AS (
    SELECT
        cm.*,
        CASE
            WHEN cm.lifetime_value >= vt.threshold THEN 'VIP'
            WHEN cm.lifetime_value > 0 THEN 'Regular'
            ELSE 'Inactive'
        END AS segment,
        CASE
            WHEN cm.last_order IS NOT NULL
            THEN cm.order_count::NUMERIC /
                GREATEST(EXTRACT(MONTH FROM AGE(cm.last_order, cm.first_order)), 1)
            ELSE 0
```

```
        END AS orders_per_month
    FROM customer_metrics cm
    CROSS JOIN vip_threshold vt
)
SELECT
    segment,
    COUNT(*) AS customer_count,
    ROUND(AVG(lifetime_value), 2) AS avg_lifetime_value,
    ROUND(AVG(order_count), 2) AS avg_orders,
    ROUND(AVG(orders_per_month), 2) AS avg_orders_per_month
FROM segmented_customers
GROUP BY segment
ORDER BY avg_lifetime_value DESC;
```

Explanation: Multiple CTEs create a clear pipeline: (1) gather metrics, (2) calculate threshold, (3) segment customers, (4) analyze segments.

ADVANCED: Recursive CTE for Hierarchical Data

Problem: Given an employee table with manager relationships, create an organizational chart showing each employee's full reporting chain from CEO down.

```
WITH RECURSIVE org_hierarchy AS (
    -- Base case: Start with the CEO
    SELECT
        emp_id,
        name,
        manager_id,
        salary,
        name AS reporting_chain,
        0 AS level
    FROM employees
    WHERE manager_id IS NULL

    UNION ALL

    -- Recursive case: Add each level of reports
    SELECT
        e.emp_id,
        e.name,
        e.manager_id,
        e.salary,
        oh.reporting_chain || ' > ' || e.name AS reporting_chain,
        oh.level + 1 AS level
```

```
    FROM employees e
    INNER JOIN org_hierarchy oh ON e.manager_id = oh.emp_id
)
SELECT
    emp_id,
    name,
    level,
    reporting_chain,
    salary
FROM org_hierarchy
ORDER BY level, name;
```

Explanation: Recursive CTEs have two parts: (1) BASE CASE defines where to start, (2) RECURSIVE CASE defines how to add the next level. Essential for hierarchical data like org charts.

---

7. WINDOW FUNCTIONS (DEEP DIVE)

PURPOSE: Window functions perform calculations across rows related to the current row without collapsing the result set. They're essential for rankings, running totals, moving averages, and comparing rows.

KEY COMPONENTS:

- OVER(): Defines the "window" of rows
- PARTITION BY: Divides rows into groups
- ORDER BY: Defines calculation order
- Frame Clause: ROWS/RANGE BETWEEN defines which rows to include

COMMON FUNCTIONS: ROW_NUMBER, RANK, DENSE_RANK, NTILE, LAG, LEAD, FIRST_VALUE, LAST_VALUE

Tables Used:

- sales_data (sale_id, employee_id, emp_name, department, sale_date, amount, region)

EASY: Basic Ranking with ROW_NUMBER

Problem: Rank all sales by amount (highest to lowest).

```
SELECT
    sale_id,
```

```
    emp_name,
    amount,
    sale_date,
    ROW_NUMBER() OVER (ORDER BY amount DESC) AS sale_rank
FROM sales_data
ORDER BY sale_rank;
```

EASY: RANK vs DENSE_RANK

Problem: Show the difference between RANK and DENSE_RANK when sales have ties.

```
SELECT
    emp_name,
    amount,
    ROW_NUMBER() OVER (ORDER BY amount DESC) AS row_num,
    RANK() OVER (ORDER BY amount DESC) AS rank,
    DENSE_RANK() OVER (ORDER BY amount DESC) AS dense_rank
FROM sales_data
ORDER BY amount DESC;
```

Explanation: ROW_NUMBER gives unique numbers. RANK gives same rank for ties but skips numbers (1,1,3,4). DENSE_RANK doesn't skip (1,1,2,3).

INTERMEDIATE: PARTITION BY for Group Rankings

Problem: Rank sales within each department separately and show percentage of department's total sales.

```
SELECT
    emp_name,
    department,
    amount,
    RANK() OVER (PARTITION BY department ORDER BY amount DESC) AS dept_rank,
    SUM(amount) OVER (PARTITION BY department) AS dept_total,
    ROUND(amount / SUM(amount) OVER (PARTITION BY department) * 100, 2) AS
pct_of_dept_sales
FROM sales_data
ORDER BY department, amount DESC;
```

Explanation: PARTITION BY creates separate windows for each department. Rankings reset for each partition.

INTERMEDIATE: LAG and LEAD for Row Comparisons

Problem: For each employee's sales over time, show previous and next sale amounts with calculations.

```
SELECT
    emp_name,
    sale_date,
    amount,
    LAG(amount) OVER (PARTITION BY employee_id ORDER BY sale_date) AS previous_sale,
    LEAD(amount) OVER (PARTITION BY employee_id ORDER BY sale_date) AS next_sale,
    amount - LAG(amount) OVER (PARTITION BY employee_id ORDER BY sale_date) AS
change_from_prev,
    ROUND(
        (amount - LAG(amount) OVER (PARTITION BY employee_id ORDER BY sale_date)) /
        NULLIF(LAG(amount) OVER (PARTITION BY employee_id ORDER BY sale_date), 0) *
100,
        2
    ) AS pct_change
FROM sales_data
ORDER BY employee_id, sale_date;
```

Explanation: LAG looks backward, LEAD looks forward. Perfect for time-series analysis and trends.

INTERMEDIATE: Running Totals and Moving Averages

Problem: Calculate running totals and 7-day moving average.

```
SELECT
    sale_date,
    amount,
    SUM(amount) OVER (ORDER BY sale_date) AS running_total,
    AVG(amount) OVER (ORDER BY sale_date) AS cumulative_avg,
    AVG(amount) OVER (
        ORDER BY sale_date
        ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
    ) AS moving_avg_7day
FROM sales_data
ORDER BY sale_date;
```

Explanation: The frame clause (ROWS BETWEEN) defines which rows to include. "6 PRECEDING AND CURRENT ROW" creates a 7-day window.

ADVANCED: NTILE for Percentile Bucketing

Problem: Divide customers into quartiles based on spending, then analyze each quartile.

```
WITH customer_spending AS (
    SELECT
        employee_id,
        emp_name,
        SUM(amount) AS total_sales
    FROM sales_data
    GROUP BY employee_id, emp_name
),
quartile_assignment AS (
    SELECT
        *,
        NTILE(4) OVER (ORDER BY total_sales) AS spending_quartile
    FROM customer_spending
)
SELECT
    spending_quartile,
    COUNT(*) AS employees_in_quartile,
    ROUND(AVG(total_sales), 2) AS avg_sales,
    ROUND(SUM(total_sales), 2) AS quartile_total
FROM quartile_assignment
GROUP BY spending_quartile
ORDER BY spending_quartile;
```

Explanation: NTILE(n) divides rows into n roughly equal groups. Perfect for segmentation analysis.

ADVANCED: FIRST_VALUE and LAST_VALUE

Problem: Compare each sale to the employee's first sale and highest sale.

```
SELECT
    emp_name,
    sale_date,
    amount,
    FIRST_VALUE(amount) OVER (
        PARTITION BY employee_id
        ORDER BY sale_date
    ) AS first_sale_amount,
    FIRST_VALUE(amount) OVER (
        PARTITION BY employee_id
```

```
    ORDER BY amount DESC
  ) AS highest_sale,
  LAST_VALUE(amount) OVER (
    PARTITION BY employee_id
    ORDER BY sale_date
    ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
  ) AS most_recent_sale
FROM sales_data
ORDER BY employee_id, sale_date;
```

Explanation: FIRST_VALUE gets the first row, LAST_VALUE gets the last. IMPORTANT: LAST_VALUE needs proper framing to see the actual last row.

---

8. COMPLEX & MULTIPLE JOINS (DEEP DIVE)

PURPOSE: Real-world analysis often requires combining 4+ tables simultaneously. Complex joins involve multiple conditions, self-joins, and understanding join order.

KEY CONCEPTS:

- Self-Joins: Joining a table to itself
- Multiple Join Conditions: AND/OR in ON clauses
- Join Order: Impacts performance
- Cross Joins: Cartesian product for combinations
- Mixed Join Types: INNER, LEFT together strategically

Tables Used:

- employees (emp_id, name, manager_id, department, hire_date)
- sales (sale_id, emp_id, customer_id, sale_date, amount)
- customers (customer_id, name, segment, region)
- products (product_id, name, category, price)
- sale_items (item_id, sale_id, product_id, quantity)

EASY: Self-Join Basics

Problem: Show each employee with their manager's name.

```
SELECT
  e.emp_id,
  e.name AS employee_name,
  e.department,
```

```
    m.name AS manager_name
FROM employees e
LEFT JOIN employees m ON e.manager_id = m.emp_id
ORDER BY e.department, e.name;
```

Explanation: Self-join uses the same table twice with different aliases. LEFT JOIN includes employees without managers (CEO).

INTERMEDIATE: Multiple Tables with Mixed Join Types

Problem: Create a report showing all employees, their sales, customer details, and product information. Include employees with no sales.

```
SELECT
    e.emp_id,
    e.name AS employee_name,
    COUNT(DISTINCT s.sale_id) AS total_sales,
    COUNT(DISTINCT s.customer_id) AS unique_customers,
    COALESCE(SUM(s.amount), 0) AS total_revenue
FROM employees e
LEFT JOIN sales s ON e.emp_id = s.emp_id
LEFT JOIN customers c ON s.customer_id = c.customer_id
LEFT JOIN sale_items si ON s.sale_id = si.sale_id
LEFT JOIN products p ON si.product_id = p.product_id
GROUP BY e.emp_id, e.name
ORDER BY total_revenue DESC;
```

Explanation: Chain multiple LEFT JOINs to preserve all employees. Each adds detail while maintaining left table rows.

INTERMEDIATE: Join with Additional Conditions

Problem: Find sales where employees sold high-value products (>$100) in their own territory region.

```
SELECT
    e.name AS employee_name,
    c.name AS customer_name,
    p.name AS product_name,
    s.amount
FROM employees e
INNER JOIN sales s ON e.emp_id = s.emp_id
INNER JOIN customers c
```

```
    ON s.customer_id = c.customer_id
    AND c.region = e.department  -- Additional condition
INNER JOIN sale_items si ON s.sale_id = si.sale_id
INNER JOIN products p
    ON si.product_id = p.product_id
    AND p.price > 100  -- Additional condition
ORDER BY s.amount DESC;
```

Explanation: Additional conditions in ON clauses filter during the join operation.

ADVANCED: Self-Join for Comparisons

Problem: Find pairs of employees in the same department where one earns 30%+ more.

```
SELECT
    e1.name AS higher_earner,
    e1.salary AS higher_salary,
    e2.name AS lower_earner,
    e2.salary AS lower_salary,
    e1.department,
    ROUND(e1.salary - e2.salary, 2) AS salary_gap
FROM employees e1
INNER JOIN employees e2
    ON e1.department = e2.department
    AND e1.emp_id < e2.emp_id  -- Avoid duplicates
    AND e1.salary > e2.salary * 1.3
ORDER BY salary_gap DESC;
```

Explanation: `e1.emp_id < e2.emp_id` ensures we don't get both (Alice, Bob) and (Bob, Alice).

ADVANCED: Cross Join for Gap Analysis

Problem: Create coverage analysis showing which employee-region combinations have sales and which are gaps.

```
WITH all_regions AS (
    SELECT DISTINCT region FROM customers
),
employee_list AS (
    SELECT emp_id, name FROM employees
),
potential_coverage AS (
```

```
    SELECT
        el.emp_id,
        el.name,
        ar.region
    FROM employee_list el
    CROSS JOIN all_regions ar
),
actual_sales AS (
    SELECT
        s.emp_id,
        c.region,
        COUNT(*) AS sales_count,
        SUM(s.amount) AS revenue
    FROM sales s
    INNER JOIN customers c ON s.customer_id = c.customer_id
    GROUP BY s.emp_id, c.region
)
SELECT
    pc.name,
    pc.region,
    CASE
        WHEN act.emp_id IS NOT NULL THEN 'Active'
        ELSE 'No Coverage'
    END AS status,
    COALESCE(act.sales_count, 0) AS sales,
    ROUND(COALESCE(act.revenue, 0), 2) AS revenue
FROM potential_coverage pc
LEFT JOIN actual_sales act
    ON pc.emp_id = act.emp_id
    AND pc.region = act.region
ORDER BY pc.name, revenue DESC;
```

Explanation: CROSS JOIN creates all possible combinations. Perfect for gap analysis and opportunity identification.

---

9. REAL-WORLD DATA ANALYST SCENARIOS

These comprehensive examples combine 5+ SQL skills to solve real business problems.

SCENARIO 1: Customer Retention Dashboard

Business Context: "We need to understand customer retention. Create a dashboard showing monthly cohorts, retention rates over time, at-risk customers, and support interaction correlation."

Tables:

- customers (customer_id, name, email, segment, signup_date)
- orders (order_id, customer_id, order_date, total_amount, status)
- support_tickets (ticket_id, customer_id, created_date, resolved_date)

Skills Used: Multiple CTEs, Window Functions (LAG, FIRST_VALUE), Complex JOINs, Date Functions, Aggregations, Subqueries

```
WITH customer_cohorts AS (
    SELECT
        customer_id,
        name,
        segment,
        DATE_TRUNC('month', signup_date) AS cohort_month
    FROM customers
),
monthly_order_activity AS (
    SELECT
        o.customer_id,
        DATE_TRUNC('month', o.order_date) AS activity_month,
        COUNT(*) AS orders_in_month,
        SUM(o.total_amount) AS revenue_in_month
    FROM orders o
    WHERE o.status = 'completed'
    GROUP BY o.customer_id, DATE_TRUNC('month', o.order_date)
),
cohort_retention AS (
    SELECT
        cc.cohort_month,
        cc.segment,
        moa.activity_month,
        EXTRACT(MONTH FROM AGE(moa.activity_month, cc.cohort_month)) AS
months_since_signup,
        COUNT(DISTINCT cc.customer_id) AS active_customers,
        SUM(moa.orders_in_month) AS total_orders
    FROM customer_cohorts cc
    INNER JOIN monthly_order_activity moa ON cc.customer_id = moa.customer_id
    GROUP BY cc.cohort_month, cc.segment, moa.activity_month
),
cohort_sizes AS (
```

```
    SELECT
        cohort_month,
        segment,
        COUNT(*) AS cohort_size
    FROM customer_cohorts
    GROUP BY cohort_month, segment
),
retention_rates AS (
    SELECT
        cr.cohort_month,
        cr.segment,
        cr.months_since_signup,
        cs.cohort_size,
        cr.active_customers,
        ROUND(cr.active_customers::NUMERIC / cs.cohort_size * 100, 2) AS retention_pct,
        LAG(cr.active_customers) OVER (
            PARTITION BY cr.cohort_month, cr.segment
            ORDER BY cr.months_since_signup
        ) AS prev_month_active
    FROM cohort_retention cr
    INNER JOIN cohort_sizes cs
        ON cr.cohort_month = cs.cohort_month
        AND cr.segment = cs.segment
),
customer_risk_analysis AS (
    SELECT
        c.customer_id,
        c.name,
        c.segment,
        MAX(o.order_date) AS last_order_date,
        COUNT(o.order_id) AS lifetime_orders,
        CURRENT_DATE - MAX(o.order_date) AS days_since_last_order,
        COUNT(CASE
            WHEN o.order_date <= c.signup_date + INTERVAL '90 days'
            THEN 1
        END) AS early_orders,
        CASE
            WHEN CURRENT_DATE - MAX(o.order_date) > 180 THEN 'Churned'
            WHEN CURRENT_DATE - MAX(o.order_date) > 90 THEN 'At Risk'
            ELSE 'Active'
        END AS risk_status
    FROM customers c
    LEFT JOIN orders o ON c.customer_id = o.customer_id
    GROUP BY c.customer_id, c.name, c.segment, c.signup_date
```

```
)
SELECT * FROM retention_rates WHERE months_since_signup <= 12
UNION ALL
SELECT * FROM customer_risk_analysis WHERE risk_status = 'At Risk';
```

SCENARIO 2: Sales Performance & Territory Optimization

Business Context: "Show each rep's performance vs peers, identify underperforming territories, find which product categories each rep should focus on, and calculate optimal territory assignments."

Tables:

- sales_reps (rep_id, name, manager_id, territory_id)
- sales (sale_id, rep_id, customer_id, product_id, sale_date, amount, stage)
- territories (territory_id, region, market_size)
- products (product_id, name, category, cost, price)

Skills Used: Self-Joins, Window Functions (RANK, NTILE, LAG), Multiple CTEs, Complex Aggregations, Correlated Subqueries

```
WITH rep_performance AS (
    SELECT
        sr.rep_id,
        sr.name,
        t.region,
        COUNT(CASE WHEN s.stage = 'Won' THEN 1 END) AS won_deals,
        SUM(CASE WHEN s.stage = 'Won' THEN s.amount ELSE 0 END) AS total_revenue,
        ROUND(
            COUNT(CASE WHEN s.stage = 'Won' THEN 1 END)::NUMERIC /
            NULLIF(COUNT(CASE WHEN s.stage IN ('Won', 'Lost') THEN 1 END), 0) * 100,
            2
        ) AS win_rate_pct
    FROM sales_reps sr
    LEFT JOIN territories t ON sr.territory_id = t.territory_id
    LEFT JOIN sales s ON sr.rep_id = s.rep_id
    GROUP BY sr.rep_id, sr.name, t.region
),
peer_benchmarks AS (
    SELECT
        rep_id,
        region,
        RANK() OVER (PARTITION BY region ORDER BY total_revenue DESC) AS region_rank,
```

```
        AVG(total_revenue) OVER (PARTITION BY region) AS region_avg_revenue,
        NTILE(100) OVER (ORDER BY total_revenue) AS revenue_percentile
    FROM rep_performance
),
category_strengths AS (
    SELECT
        s.rep_id,
        p.category,
        SUM(s.amount) AS category_revenue,
        RANK() OVER (PARTITION BY s.rep_id ORDER BY SUM(s.amount) DESC) AS
category_rank
    FROM sales s
    INNER JOIN products p ON s.product_id = p.product_id
    WHERE s.stage = 'Won'
    GROUP BY s.rep_id, p.category
)
SELECT
    rp.rep_id,
    rp.name,
    rp.region,
    rp.won_deals,
    ROUND(rp.total_revenue, 2) AS revenue,
    rp.win_rate_pct,
    pb.region_rank,
    ROUND(pb.region_avg_revenue, 2) AS region_avg,
    pb.revenue_percentile,
    (SELECT category
     FROM category_strengths cs
     WHERE cs.rep_id = rp.rep_id AND cs.category_rank = 1
    ) AS top_category,
    CASE
        WHEN pb.revenue_percentile >= 75 THEN 'Star Performer'
        WHEN pb.revenue_percentile >= 50 THEN 'Solid Contributor'
        ELSE 'Needs Support'
    END AS performance_category
FROM rep_performance rp
INNER JOIN peer_benchmarks pb ON rp.rep_id = pb.rep_id
ORDER BY rp.total_revenue DESC;
```

SCENARIO 3: Product Portfolio & Inventory Optimization

Business Context: "We need a data-driven inventory strategy. Show product performance trends, identify fast vs slow movers, find products bought together, calculate optimal reorder points, and flag stockout/overstock risks."

Tables:

- products (product_id, name, category, cost, price, current_stock)
- sales_transactions (trans_id, trans_date, customer_id)
- transaction_items (item_id, trans_id, product_id, quantity)

Skills Used: Self-Joins (product pairs), Window Functions (LAG, moving averages), CTEs, Date Math, Advanced Aggregations

```
WITH product_sales_metrics AS (
    SELECT
        p.product_id,
        p.name,
        p.category,
        p.current_stock,
        SUM(ti.quantity) AS units_sold,
        SUM(CASE
            WHEN st.trans_date >= CURRENT_DATE - INTERVAL '30 days'
            THEN ti.quantity
        END) AS units_last_30,
        SUM(CASE
            WHEN st.trans_date BETWEEN CURRENT_DATE - INTERVAL '60 days'
                                   AND CURRENT_DATE - INTERVAL '31 days'
            THEN ti.quantity
        END) AS units_prev_30,
        ROUND(
            SUM(ti.quantity)::NUMERIC /
            NULLIF(EXTRACT(DAY FROM MAX(st.trans_date) - MIN(st.trans_date)), 0),
            2
        ) AS avg_daily_sales
    FROM products p
    LEFT JOIN transaction_items ti ON p.product_id = ti.product_id
    LEFT JOIN sales_transactions st ON ti.trans_id = st.trans_id
    GROUP BY p.product_id, p.name, p.category, p.current_stock
),
sales_velocity AS (
    SELECT
        *,
        CASE
            WHEN avg_daily_sales > 0
            THEN ROUND(current_stock / avg_daily_sales, 1)
```

```
            ELSE 999
        END AS days_of_stock,
        CASE
            WHEN units_prev_30 > 0 AND units_last_30 > units_prev_30 * 1.2
                THEN 'Accelerating'
            WHEN units_last_30 < units_prev_30 * 0.8
                THEN 'Declining'
            ELSE 'Stable'
        END AS velocity_trend,
        CASE
            WHEN avg_daily_sales >= 10 THEN 'Fast Mover'
            WHEN avg_daily_sales >= 3 THEN 'Moderate'
            ELSE 'Slow'
        END AS velocity_class
    FROM product_sales_metrics
),
inventory_risk AS (
    SELECT
        *,
        CASE
            WHEN days_of_stock < 7 AND velocity_class = 'Fast Mover'
                THEN 'Critical Stockout Risk'
            WHEN days_of_stock < 14 AND velocity_class IN ('Fast Mover', 'Moderate')
                THEN 'Stockout Risk'
            WHEN days_of_stock > 90 AND velocity_class = 'Slow'
                THEN 'Overstock'
            ELSE 'Normal'
        END AS inventory_status,
        CASE
            WHEN days_of_stock < 30
            THEN ROUND((30 - days_of_stock) * avg_daily_sales, 0)
            ELSE 0
        END AS recommended_reorder_qty
    FROM sales_velocity
),
product_pairs AS (
    SELECT
        ti1.product_id AS product_a,
        ti2.product_id AS product_b,
        COUNT(DISTINCT ti1.trans_id) AS times_bought_together
    FROM transaction_items ti1
    INNER JOIN transaction_items ti2
        ON ti1.trans_id = ti2.trans_id
        AND ti1.product_id < ti2.product_id
```

```
    GROUP BY ti1.product_id, ti2.product_id
    HAVING COUNT(DISTINCT ti1.trans_id) >= 5
)
SELECT
    ir.product_id,
    ir.name,
    ir.category,
    ir.current_stock,
    ir.velocity_class,
    ir.velocity_trend,
    ROUND(ir.avg_daily_sales, 2) AS daily_sales_rate,
    ir.days_of_stock,
    ir.inventory_status,
    ir.recommended_reorder_qty,
    (SELECT STRING_AGG(p.name, ', ')
     FROM product_pairs pp
     INNER JOIN products p ON pp.product_b = p.product_id
     WHERE pp.product_a = ir.product_id
     LIMIT 3
    ) AS frequently_bought_with,
    CASE
        WHEN ir.inventory_status = 'Critical Stockout Risk'
            THEN 'URGENT: Reorder Immediately'
        WHEN ir.inventory_status = 'Overstock' AND ir.velocity_trend = 'Declining'
            THEN 'Consider Clearance'
        WHEN ir.velocity_trend = 'Accelerating' AND ir.days_of_stock < 30
            THEN 'Increase Order Quantity'
        ELSE 'Monitor'
    END AS recommended_action
FROM inventory_risk ir
ORDER BY
    CASE ir.inventory_status
        WHEN 'Critical Stockout Risk' THEN 1
        WHEN 'Stockout Risk' THEN 2
        ELSE 3
    END;
```

PRACTICE TIPS

Building Your Skills:

1. Start simple, build complexity - Write each CTE separately and test before adding more

2. Use EXPLAIN ANALYZE for performance insights on large datasets
3. Comment your CTEs to explain what each calculates
4. Test with LIMIT 100 during development to speed up iterations
5. Practice translating business questions into SQL components

Common Pitfalls:

- Division by Zero: Always use NULLIF(column, 0) in denominators
- NULL Comparisons: NULL = NULL is FALSE; use IS NULL
- Window Function Frames: Default may not work for LAST_VALUE
- GROUP BY Requirements: Every non-aggregated column must be in GROUP BY

Interview Preparation: Be ready to explain:

1. Why you chose a particular approach
2. Alternative methods you considered
3. Performance implications
4. How you'd modify for edge cases

END OF GUIDE

Remember: Real data analyst work involves translating vague business questions into specific SQL queries, cleaning messy data, and communicating findings clearly. Master these patterns and you'll handle any analytical challenge!