100 Essential SQL Queries for Data Analysts

This cheat sheet provides a quick reference to common SQL queries and concepts used in data analysis, from fundamental commands to advanced techniques.

Section 1: Basic Queries (1-10)

Use these to select and filter data from a table.

1. **Select all columns from a table.**
   ```
   SELECT * FROM employees;
   ```

2. **Select specific columns.**
   ```
   SELECT first_name, last_name, hire_date FROM employees;
   ```

3. **Count all rows.**
   ```
   SELECT COUNT(*) FROM employees;
   ```

4. **Count the number of non-null values in a column.**
   ```
   SELECT COUNT(salary) FROM employees;
   ```

5. **Find unique values in a column.**
   ```
   SELECT DISTINCT department FROM employees;
   ```

6. **Filter data using a WHERE clause.**
   ```
   SELECT * FROM employees WHERE department = 'Sales';
   ```

7. **Filter with multiple conditions using AND.**
   ```
   SELECT * FROM employees WHERE department = 'Sales' AND salary >
   60000;
   ```

8. **Filter with multiple conditions using OR.**
   ```
   SELECT * FROM employees WHERE department = 'Sales' OR department =
   'Marketing';
   ```

9. **Combine AND and OR with parentheses for complex logic.**
   ```
   SELECT * FROM employees WHERE (department = 'Sales' OR department
   = 'Marketing') AND salary > 70000;
   ```

10. **Exclude a value using NOT.**
    ```
    SELECT * FROM employees WHERE NOT department = 'IT';
    ```

Section 2: Filtering & Pattern Matching (11-20)

Advanced filtering using IN, BETWEEN, and LIKE.

1. **Filter by a list of values using IN.**
   ```
   SELECT * FROM employees WHERE department IN ('Sales', 'Marketing',
   'IT');
   ```

2. **Exclude a list of values using NOT IN.**
   ```
   SELECT * FROM employees WHERE department NOT IN ('Sales',
   'Marketing');
   ```

3. **Filter values within a range using BETWEEN.**
   ```
   SELECT * FROM employees WHERE salary BETWEEN 50000 AND 75000;
   ```

4. **Find rows where a string contains a substring (case-insensitive).**
   ```
   SELECT * FROM employees WHERE first_name ILIKE '%jo%';
   ```

5. **Find rows where a string starts with a specific pattern.**
   ```
   SELECT * FROM employees WHERE last_name LIKE 'Smi%';
   ```

6. **Find rows where a string ends with a specific pattern.**
   ```
   SELECT * FROM employees WHERE email LIKE '%@gmail.com';
   ```

7. **Find rows with a pattern at a specific position.**
   ```
   SELECT * FROM employees WHERE first_name LIKE '_a%';
   ```

8. **Find rows with NULL values.**
   ```
   SELECT * FROM employees WHERE manager_id IS NULL;
   ```

9. **Find rows with non-NULL values.**
   ```
   SELECT * FROM employees WHERE manager_id IS NOT NULL;
   ```

10. **Filter by date range.**
    ```
    SELECT * FROM employees WHERE hire_date BETWEEN '2023-01-01' AND
    '2023-12-31';
    ```

Section 3: Sorting & Limiting (21-30)

Control the order of your results and limit the number of rows.

1. **Sort results in ascending order.**
   ```
   SELECT * FROM employees ORDER BY last_name ASC;
   ```

2. **Sort results in descending order.**
   ```
   SELECT * FROM employees ORDER BY salary DESC;
   ```

3. **Sort by multiple columns.**
   ```
   SELECT * FROM employees ORDER BY department ASC, salary DESC;
   ```

4. **Get the top 10 highest paid employees.**
   ```
   SELECT * FROM employees ORDER BY salary DESC LIMIT 10;
   ```

5. **Get the latest 5 hired employees.**
   ```
   SELECT * FROM employees ORDER BY hire_date DESC LIMIT 5;
   ```

6. **Combine DISTINCT and ORDER BY.**
   ```
   SELECT DISTINCT department FROM employees ORDER BY department;
   ```

7. **Fetch a specific range of rows (e.g., rows 11-20).**
   ```
   SELECT * FROM employees ORDER BY employee_id OFFSET 10 LIMIT 10;
   ```

8. **Get the employee with the highest salary.**
   ```
   SELECT * FROM employees ORDER BY salary DESC LIMIT 1;
   ```

9. **Get the second-highest salary.**
   ```
   SELECT DISTINCT salary FROM employees ORDER BY salary DESC LIMIT 1
   OFFSET 1;
   ```

10. **Order by a calculated field.**
    ```
    SELECT first_name, last_name, (salary * 0.1) AS bonus FROM
    employees ORDER BY bonus DESC;
    ```

Section 4: Aggregation & Grouping (31-40)

Summarize and analyze data with aggregate functions.

1. **Count rows for each group.**
   ```
   SELECT department, COUNT(*) FROM employees GROUP BY department;
   ```

2. **Calculate the average salary per department.**
   ```
   SELECT department, AVG(salary) FROM employees GROUP BY department;
   ```

3. **Find the total salary expenditure for each department.**
   ```
   SELECT department, SUM(salary) FROM employees GROUP BY department;
   ```

4. **Find the maximum salary in each department.**
   ```
   SELECT department, MAX(salary) FROM employees GROUP BY department;
   ```

5. **Find the minimum salary in each department.**
   ```
   SELECT department, MIN(salary) FROM employees GROUP BY department;
   ```

6. **Filter groups using HAVING.**
   ```
   SELECT department, AVG(salary) FROM employees GROUP BY department
   HAVING AVG(salary) > 65000;
   ```

7. **Count employees in departments with more than 5 employees.**
```
SELECT department, COUNT(*) FROM employees GROUP BY department
HAVING COUNT(*) > 5;
```

8. **Filter and then group.**
```
SELECT department, AVG(salary) FROM employees WHERE hire_date >
'2022-01-01' GROUP BY department;
```

9. **Group by multiple columns.**
```
SELECT department, job_title, AVG(salary) FROM employees GROUP BY
department, job_title;
```

10. **Find the first and last name of the highest earner in each department.**
```
SELECT department, MAX(salary) FROM employees GROUP BY department;
```

Section 5: Joins (41-60)

Combine data from multiple tables.

1. **INNER JOIN: Return matching rows from both tables.**
```
SELECT e.first_name, d.department_name
FROM employees e
INNER JOIN departments d ON e.department_id = d.department_id;
```

2. **LEFT JOIN: Return all rows from the left table, and the matched rows from the right table.**
```
SELECT e.first_name, d.department_name
FROM employees e
LEFT JOIN departments d ON e.department_id = d.department_id;
```

3. **LEFT JOIN to find rows in the left table with no match in the right.**
```
SELECT e.first_name FROM employees e LEFT JOIN departments d ON
e.department_id = d.department_id WHERE d.department_id IS NULL;
```

4. **RIGHT JOIN: Return all rows from the right table, and the matched rows from the left table.**
```
SELECT e.first_name, d.department_name
FROM employees e
RIGHT JOIN departments d ON e.department_id = d.department_id;
```

5. **FULL OUTER JOIN: Return all rows when there is a match in either table.**
```
SELECT e.first_name, d.department_name
FROM employees e
FULL OUTER JOIN departments d ON e.department_id =
d.department_id;
```

6. **Join three tables.**

```
SELECT e.first_name, p.project_name, d.department_name
FROM employees e
JOIN projects p ON e.employee_id = p.employee_id
JOIN departments d ON e.department_id = d.department_id;
```

7. **Self-Join: Join a table to itself to compare rows within the same table.**
```
SELECT e.first_name AS employee, m.first_name AS manager
FROM employees e
JOIN employees m ON e.manager_id = m.employee_id;
```

8. **Join a table on a non-key column.**
```
SELECT o.order_id, c.customer_name
FROM orders o
JOIN customers c ON o.customer_zip_code = c.customer_zip_code;
```

9. **Cross Join: Return the Cartesian product of the two tables.**
```
SELECT * FROM employees CROSS JOIN departments;
```

10. **Left Join with filtering (similar to #43, but a common pattern).**
```
SELECT * FROM products p LEFT JOIN sales s ON p.product_id =
s.product_id WHERE s.sale_id IS NULL;
```

11. **Find employees who have not been assigned to a project.**
```
SELECT e.first_name, e.last_name FROM employees e LEFT JOIN
employee_projects ep ON e.employee_id = ep.employee_id WHERE
ep.project_id IS NULL;
```

12. **Find projects without any employees assigned.**
```
SELECT p.project_name FROM projects p LEFT JOIN employee_projects
ep ON p.project_id = ep.project_id WHERE ep.employee_id IS NULL;
```

13. **Find the total salary of each department, including departments with no employees.**
```
SELECT d.department_name, SUM(e.salary) AS total_salary FROM
departments d LEFT JOIN employees e ON d.department_id =
e.department_id GROUP BY d.department_name;
```

14. **Find the number of employees in each department, including those with zero.**
```
SELECT d.department_name, COUNT(e.employee_id) AS num_employees
FROM departments d LEFT JOIN employees e ON d.department_id =
e.department_id GROUP BY d.department_name;
```

15. **Join based on multiple conditions.**
```
SELECT o.*, c.customer_name FROM orders o JOIN customers c ON
o.customer_id = c.customer_id AND o.order_date =
c.last_purchase_date;
```

16. **Find employees and their managers, and their managers' departments.**
```
SELECT e.first_name, m.first_name AS manager_name,
```

```
d.department_name FROM employees e JOIN employees m ON
e.manager_id = m.employee_id JOIN departments d ON m.department_id
= d.department_id;
```

17. **Join on a non-equi condition.**
```
SELECT e.employee_id, h.salary_grade FROM employees e JOIN
salary_grades h ON e.salary BETWEEN h.min_salary AND h.max_salary;
```

18. **Filter a joined table using WHERE.**
```
SELECT e.first_name, d.department_name FROM employees e JOIN
departments d ON e.department_id = d.department_id WHERE
d.location_city = 'New York';
```

19. **Perform an aggregation on a joined table.**
```
SELECT d.department_name, AVG(e.salary) FROM employees e JOIN
departments d ON e.department_id = d.department_id GROUP BY
d.department_name;
```

20. **Join on a common column with different names.**
```
SELECT a.order_id, b.item_name FROM table_a a JOIN table_b b ON
a.product_id = b.item_id;
```

Section 6: Subqueries (61-75)

Use a query within another query.

1. **Subquery in the WHERE clause (single-value).**
```
SELECT * FROM employees WHERE salary > (SELECT AVG(salary) FROM
employees);
```

2. **Subquery in the WHERE clause (multiple-values).**
```
SELECT * FROM employees WHERE department_id IN (SELECT
department_id FROM departments WHERE location_city = 'San
Francisco');
```

3. **Subquery in the FROM clause (derived table).**
```
SELECT department_name, avg_salary
FROM (SELECT department_id, AVG(salary) AS avg_salary FROM
employees GROUP BY department_id) AS avg_salaries
JOIN departments ON avg_salaries.department_id =
departments.department_id;
```

4. **Subquery in the SELECT clause (scalar subquery).**
```
SELECT first_name, salary, (SELECT AVG(salary) FROM employees) AS
company_avg_salary FROM employees;
```

5. **Correlated Subquery: Find employees whose salary is greater than the average**

salary of their own department.

```
SELECT * FROM employees e WHERE salary > (SELECT AVG(salary) FROM
employees WHERE department_id = e.department_id);
```

6. **Using EXISTS to check for existence of rows.**

```
SELECT department_name FROM departments d WHERE EXISTS (SELECT 1
FROM employees WHERE department_id = d.department_id AND salary >
100000);
```

7. **Using NOT EXISTS to find non-matching rows.**

```
SELECT department_name FROM departments d WHERE NOT EXISTS (SELECT
1 FROM employees WHERE department_id = d.department_id);
```

8. **Find departments with at least one employee.**

```
SELECT * FROM departments WHERE department_id IN (SELECT DISTINCT
department_id FROM employees);
```

9. **Find employees who have placed an order.**

```
SELECT * FROM employees e WHERE EXISTS (SELECT 1 FROM orders o
WHERE o.employee_id = e.employee_id);
```

10. **Find the total salary for each department using a subquery.**

```
SELECT department_name, (SELECT SUM(salary) FROM employees WHERE
department_id = d.department_id) AS total_department_salary FROM
departments d;
```

11. **Select the name of the department with the highest average salary.**

```
SELECT department_name FROM departments WHERE department_id =
(SELECT department_id FROM employees GROUP BY department_id ORDER
BY AVG(salary) DESC LIMIT 1);
```

12. **Find customers who have ordered a specific product.**

```
SELECT customer_name FROM customers WHERE customer_id IN (SELECT
customer_id FROM orders WHERE product_id = 123);
```

13. **Find the number of employees who earn more than the average salary.**

```
SELECT COUNT(*) FROM employees WHERE salary > (SELECT AVG(salary)
FROM employees);
```

14. **Get a list of customers who have not placed an order.**

```
SELECT customer_name FROM customers WHERE customer_id NOT IN
(SELECT DISTINCT customer_id FROM orders);
```

15. **Find employees with a salary less than the average of their job title.**

```
SELECT e.first_name, e.salary, e.job_title FROM employees e WHERE
e.salary < (SELECT AVG(salary) FROM employees WHERE job_title =
e.job_title);
```

Section 7: Window Functions (76-85)

Perform calculations across a set of table rows that are related to the current row.

1. **ROW_NUMBER(): Assign a unique sequential integer to each row within a partition.**
   ```
   SELECT first_name, department, salary, ROW_NUMBER() OVER
   (PARTITION BY department ORDER BY salary DESC) AS rn FROM
   employees;
   ```

2. **RANK(): Assign a rank to each row within a partition, with ties receiving the same rank and a gap in the sequence.**
   ```
   SELECT first_name, department, salary, RANK() OVER (PARTITION BY
   department ORDER BY salary DESC) AS rank FROM employees;
   ```

3. **DENSE_RANK(): Assign a rank to each row within a partition, with no gaps in the sequence.**
   ```
   SELECT first_name, department, salary, DENSE_RANK() OVER
   (PARTITION BY department ORDER BY salary DESC) AS dense_rank FROM
   employees;
   ```

4. **NTILE(n): Distribute rows into a specified number of groups (n).**
   ```
   SELECT first_name, salary, NTILE(4) OVER (ORDER BY salary DESC) AS
   quartile FROM employees;
   ```

5. **LEAD(): Access data from a subsequent row.**
   ```
   SELECT order_date, total_amount, LEAD(total_amount, 1) OVER (ORDER
   BY order_date) AS next_order_amount FROM orders;
   ```

6. **LAG(): Access data from a previous row.**
   ```
   SELECT order_date, total_amount, LAG(total_amount, 1, 0) OVER
   (ORDER BY order_date) AS previous_order_amount FROM orders;
   ```

7. **Calculate a running total.**
   ```
   SELECT order_date, total_amount, SUM(total_amount) OVER (ORDER BY
   order_date ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS
   running_total FROM orders;
   ```

8. **Calculate a moving average.**
   ```
   SELECT order_date, total_amount, AVG(total_amount) OVER (ORDER BY
   order_date ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS moving_avg
   FROM orders;
   ```

9. **Find the top 3 employees in each department using RANK().**
   ```
   SELECT * FROM (
     SELECT first_name, department, salary, RANK() OVER (PARTITION BY
   department ORDER BY salary DESC) AS rn FROM employees
   ) AS ranked_employees
   ```

```
WHERE rn <= 3;
```

10. **Calculate the percentage of total sales for each product.**
```
SELECT product_id, SUM(sales) as product_sales, SUM(SUM(sales))
OVER () as total_sales, (SUM(sales) / SUM(SUM(sales)) OVER ()) *
100 as percentage_of_total FROM sales GROUP BY product_id;
```

Section 8: Common Table Expressions (CTEs) & Data Manipulation (86-100)

Organize complex queries and modify data.

1. **Use a CTE to simplify a query.**
```
WITH department_avg AS (
  SELECT department_id, AVG(salary) AS avg_dept_salary FROM
employees GROUP BY department_id
)
SELECT e.first_name, e.salary, da.avg_dept_salary FROM employees e
JOIN department_avg da ON e.department_id = da.department_id WHERE
e.salary > da.avg_dept_salary;
```

2. **INSERT INTO: Add a new row to a table.**
```
INSERT INTO employees (first_name, last_name, salary) VALUES
('John', 'Doe', 60000);
```

3. **UPDATE: Modify existing data.**
```
UPDATE employees SET salary = salary * 1.05 WHERE department =
'IT';
```

4. **DELETE: Remove rows from a table.**
```
DELETE FROM employees WHERE employee_id = 101;
```

5. **TRUNCATE TABLE: Quickly delete all rows from a table.**
```
TRUNCATE TABLE old_data;
```

6. **UNION: Combine the result sets of two or more SELECT statements (removes duplicates).**
```
SELECT first_name FROM employees UNION SELECT first_name FROM
customers;
```

7. **UNION ALL: Combine result sets without removing duplicates.**
```
SELECT first_name FROM employees UNION ALL SELECT first_name FROM
customers;
```

8. **CASE statement for conditional logic.**
```
SELECT first_name, salary,
```

```
  CASE
    WHEN salary > 100000 THEN 'High Earner'
    WHEN salary BETWEEN 50000 AND 100000 THEN 'Mid-Range'
    ELSE 'Junior'
  END AS salary_level
FROM employees;
```

9. **Pivot data using CASE and GROUP BY.**
```
SELECT
  department,
  COUNT(CASE WHEN salary > 70000 THEN 1 END) AS high_salary_count,
  COUNT(CASE WHEN salary <= 70000 THEN 1 END) AS low_salary_count
FROM employees
GROUP BY department;
```

10. **CAST: Convert one data type to another.**
```
SELECT CAST(order_date AS DATE) FROM orders;
```

11. **COALESCE: Return the first non-null expression in a list.**
```
SELECT COALESCE(email, 'No Email Provided') FROM employees;
```

12. **NULLIF: Returns NULL if the two arguments are equal.**
```
SELECT NULLIF(salary, 0) FROM employees;
```

13. **CTE for finding recursive relationships.**
```
WITH RECURSIVE subordinates AS (
  SELECT employee_id, manager_id FROM employees WHERE employee_id
= 1
  UNION ALL
  SELECT e.employee_id, e.manager_id FROM employees e JOIN
subordinates s ON e.manager_id = s.employee_id
)
SELECT * FROM subordinates;
```

14. **GROUPING SETS: Perform grouping on multiple sets of columns.**
```
SELECT department, job_title, SUM(salary) FROM employees GROUP BY
GROUPING SETS ((department), (job_title), ());
```

15. **ROLLUP: Generate subtotals and a grand total.** sql SELECT department, job_title, SUM(salary) FROM employees GROUP BY ROLLUP(department, job_title);