SQL for Data Analysts

Essential Queries & Commands Cheat Sheet

■ BASIC QUERIES

SELECT - Retrieve Data

```
SELECT column1, column2 FROM table_name;
SELECT * FROM customers; -- All columns
SELECT DISTINCT city FROM customers; -- Unique values
```

WHERE - Filter Results

```
SELECT * FROM orders WHERE status = 'completed';
SELECT * FROM products WHERE price > 100;
SELECT * FROM users WHERE created_at >= '2024-01-01';
```

ORDER BY - Sort Results

```
SELECT * FROM products ORDER BY price ASC;
SELECT * FROM products ORDER BY price DESC;
SELECT * FROM sales ORDER BY date DESC, amount DESC;
```

LIMIT - Restrict Rows

```
SELECT * FROM customers LIMIT 10;
SELECT * FROM orders ORDER BY total DESC LIMIT 5; -- Top 5
```

■ FILTERING & CONDITIONS

Comparison Operators

```
= Equal to <> Not equal to
> Greater than >= Greater or equal
< Less than <= Less or equal
```

Logical Operators

```
SELECT * FROM orders WHERE status = 'shipped' AND total > 100;
SELECT * FROM products WHERE category = 'Electronics' OR category = 'Books';
SELECT * FROM users WHERE NOT country = 'USA';
```

IN, BETWEEN, LIKE

```
SELECT * FROM orders WHERE status IN ('pending', 'processing');
SELECT * FROM products WHERE price BETWEEN 50 AND 200;
SELECT * FROM customers WHERE name LIKE 'John%'; -- Starts with John
SELECT * FROM customers WHERE email LIKE '%@gmail.com'; -- Ends with
```

NULL Handling

```
SELECT * FROM customers WHERE phone IS NULL;
SELECT * FROM orders WHERE shipped_date IS NOT NULL;
SELECT COALESCE(phone, 'No phone') FROM customers; -- Default value
```

■ AGGREGATION FUNCTIONS

Common Aggregates

```
SELECT COUNT(*) FROM orders; -- Count all rows
SELECT COUNT(DISTINCT customer_id) FROM orders; -- Count unique
SELECT SUM(amount) FROM transactions;
SELECT AVG(price) FROM products;
SELECT MIN(price), MAX(price) FROM products;
```

GROUP BY - Aggregate by Category

```
SELECT category, COUNT(*) as product_count
FROM products
GROUP BY category;

SELECT customer_id, SUM(total) as total_spent
FROM orders
GROUP BY customer_id
ORDER BY total_spent DESC;
```

HAVING - Filter Aggregated Results

```
SELECT customer_id, COUNT(*) as order_count
FROM orders
GROUP BY customer_id
HAVING COUNT(*) > 5; -- Customers with more than 5 orders
```

■ JOINS

INNER JOIN - Matching rows only

```
SELECT orders.id, customers.name, orders.total
FROM orders
INNER JOIN customers ON orders.customer_id = customers.id;
```

LEFT JOIN - All from left + matches

```
SELECT customers.name, orders.id
FROM customers
LEFT JOIN orders ON customers.id = orders.customer_id;
-- Includes customers with no orders (NULL for order columns)
```

RIGHT JOIN - All from right + matches

```
SELECT customers.name, orders.id
FROM customers
RIGHT JOIN orders ON customers.id = orders.customer_id;
```

Multiple Joins

```
SELECT o.id, c.name, p.product_name, oi.quantity
FROM orders o
JOIN customers c ON o.customer_id = c.id
JOIN order_items oi ON o.id = oi.order_id
JOIN products p ON oi.product_id = p.id;
```

■ SUBQUERIES & CTEs

Subquery in WHERE

```
SELECT * FROM products
WHERE price > (SELECT AVG(price) FROM products);

SELECT * FROM customers
WHERE id IN (SELECT customer_id FROM orders WHERE total > 1000);
```

Common Table Expression (CTE)

```
WITH high_value_customers AS (
SELECT customer_id, SUM(total) as total_spent
FROM orders
GROUP BY customer_id
HAVING SUM(total) > 10000
)
SELECT c.name, hvc.total_spent
FROM high_value_customers hvc
JOIN customers c ON hvc.customer_id = c.id;
```

■ WINDOW FUNCTIONS

ROW_NUMBER, RANK, DENSE_RANK

```
SELECT name, department, salary,
ROW_NUMBER() OVER (ORDER BY salary DESC) as row_num,
RANK() OVER (ORDER BY salary DESC) as rank,
DENSE_RANK() OVER (ORDER BY salary DESC) as dense_rank
FROM employees;
```

Partition By - Window within groups

```
SELECT name, department, salary,
RANK() OVER (PARTITION BY department ORDER BY salary DESC) as dept_rank
FROM employees;
-- Ranks employees within each department
```

Running Totals & Moving Averages

```
SELECT date, amount,
SUM(amount) OVER (ORDER BY date) as running_total,
```

```
AVG(amount) OVER (ORDER BY date ROWS 6 PRECEDING) as 7_day_avg
FROM daily_sales;
```

■ DATE FUNCTIONS

Common Date Operations

```
SELECT CURRENT_DATE; -- Today's date
SELECT CURRENT_TIMESTAMP; -- Current datetime
SELECT DATE_PART('year', order_date); -- Extract year
SELECT DATE_TRUNC('month', order_date); -- Truncate to month
SELECT order_date + INTERVAL '30 days'; -- Add 30 days
SELECT AGE(end_date, start_date); -- Difference
```

Group by Time Periods

```
-- Monthly sales
SELECT DATE_TRUNC('month', order_date) as month,
SUM(total) as monthly_sales
FROM orders
GROUP BY DATE_TRUNC('month', order_date)
ORDER BY month;
```

■ STRING FUNCTIONS

```
SELECT UPPER(name), LOWER(email) FROM customers;
SELECT CONCAT(first_name, ' ', last_name) as full_name FROM users;
SELECT SUBSTRING(phone, 1, 3) as area_code FROM contacts;
SELECT TRIM(name) FROM products; -- Remove whitespace
SELECT LENGTH(description) FROM products;
SELECT REPLACE(text, 'old', 'new') FROM documents;
```

■ CASE STATEMENTS

```
SELECT name, price,
CASE
WHEN price < 50 THEN 'Budget'
WHEN price < 200 THEN 'Mid-range'
ELSE 'Premium'
END as price_tier
FROM products;

-- Conditional aggregation
SELECT
COUNT(CASE WHEN status = 'completed' THEN 1 END) as completed,
COUNT(CASE WHEN status = 'pending' THEN 1 END) as pending
FROM orders;
```

✏️ DATA MODIFICATION

INSERT

```
INSERT INTO customers (name, email) VALUES ('John', 'john@email.com');
INSERT INTO orders (customer_id, total)
SELECT id, 0 FROM customers WHERE status = 'new';
```

UPDATE

```
UPDATE products SET price = price * 1.1 WHERE category = 'Electronics';
UPDATE orders SET status = 'shipped' WHERE id = 123;
```

DELETE

```
DELETE FROM orders WHERE status = 'cancelled';
DELETE FROM logs WHERE created_at < '2023-01-01';
```

⚡ PERFORMANCE TIPS

Tip	Why
Use SELECT columns instead of SELECT *	Reduces data transfer
Add indexes on WHERE columns	Speeds up filtering
Use EXPLAIN ANALYZE	Understand query performance
Avoid functions on indexed columns	Prevents index usage
Use LIMIT for large datasets	Reduces memory usage
Filter early with WHERE	Reduces rows to process

Created by Ayoolumi Melehon | ayofemimelehon.com
Data Analyst & AI Developer | Free to use and share