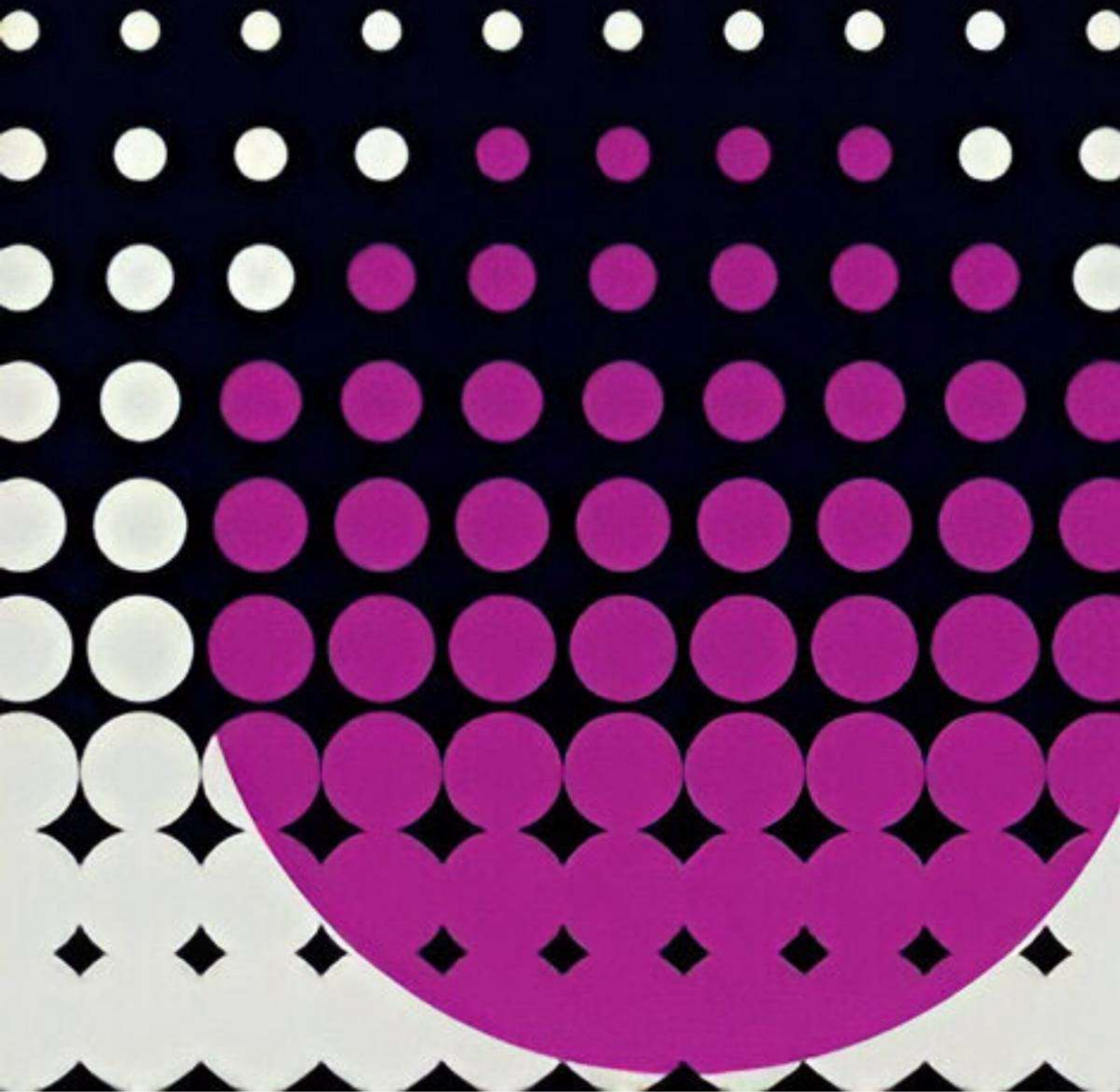
Advanced SQL Techniques CTEs, Subqueries, and More

With Code Examples



save for later

Common Table Expressions (CTEs)

CTEs are temporary named result sets that exist within the scope of a single SQL statement. They simplify complex queries by breaking them into smaller, more manageable parts.

```
WITH sales_summary AS (
    SELECT
        product_id,
        SUM(quantity) AS total_quantity,
        SUM(price * quantity) AS total_revenue
    FROM sales
    GROUP BY product_id
)
SELECT
    p.product_name,
    s.total_quantity,
    s.total_revenue
FROM products p
JOIN sales_summary s ON p.product_id =
s.product_id
ORDER BY s.total_revenue DESC
LIMIT 10;
```

Subqueries

Subqueries are nested queries within a larger SQL statement. They can be used in various parts of a query, such as SELECT, FROM, WHERE, and HAVING clauses.

```
SELECT
    employee_name,
    salary
FROM employees
WHERE salary > (
    SELECT AVG(salary)
    FROM employees
    WHERE department = 'Sales'
)
ORDER BY salary DESC;
```

save for later

Self Joins

Self joins are used when a table needs to be joined with itself, typically to compare rows within the same table or to establish hierarchical relationships.

```
SELECT
    e.employee_name AS employee,
    m.employee_name AS manager
FROM employees e
LEFT JOIN employees m ON e.manager_id =
m.employee_id
ORDER BY e.employee_name;
```

Swipe next

Window Functions

Window functions perform calculations across a set of table rows that are related to the current row, allowing for complex analytical queries.

```
SELECT
    employee_name,
    department,
    salary,
    AVG(salary) OVER (PARTITION BY department)
AS dept_avg_salary,
    salary - AVG(salary) OVER (PARTITION BY
department) AS salary_diff_from_avg
FROM employees
ORDER BY department, salary DESC;
```

Unions

UNION combines the result sets of two or more SELECT statements, removing duplicate rows by default. UNION ALL retains all rows, including duplicates.

```
SELECT product_name, 'In Stock' AS status
FROM products
WHERE stock_quantity > 0

UNION

SELECT product_name, 'Out of Stock' AS status
FROM products
WHERE stock_quantity = 0

ORDER BY product_name;
```

follow for more

Date Manipulation

SQL provides various functions to work with dates, allowing for complex date-based calculations and filtering.

```
SELECT
    order_id,
    order_date,
    delivery_date,
    DATEDIFF(delivery_date, order_date) AS
days_to_deliver,
    DATE_ADD(order_date, INTERVAL 7 DAY) AS
expected_delivery,
    CASE
        WHEN delivery_date <=
DATE_ADD(order_date, INTERVAL 7 DAY) THEN 'On
Time'
        ELSE 'Delayed'
    END AS delivery_status
FROM orders
WHERE YEAR(order_date) = YEAR(CURDATE())
ORDER BY order_date;
```

Swipe next ⟶

save for later

Pivoting Techniques

Pivoting transforms rows into columns, useful for creating summary reports or transforming data for analysis.

```
SELECT
    product_category,
    SUM(CASE WHEN MONTH(order_date) = 1 THEN
total_amount ELSE 0 END) AS Jan_sales,
    SUM(CASE WHEN MONTH(order_date) = 2 THEN
total_amount ELSE 0 END) AS Feb_sales,
    SUM(CASE WHEN MONTH(order_date) = 3 THEN
total_amount ELSE 0 END) AS Mar_sales
FROM sales
WHERE YEAR(order_date) = YEAR(CURDATE())
GROUP BY product_category
ORDER BY product_category;
```

follow for more

Unpivoting Techniques

Unpivoting converts columns into rows, useful for normalizing data or preparing it for analysis.

```sql
SELECT
    product_id,
    'Jan_sales' AS month,
    Jan_sales AS sales_amount
FROM monthly_sales
UNION ALL
SELECT
    product_id,
    'Feb_sales' AS month,
    Feb_sales AS sales_amount
FROM monthly_sales
UNION ALL
SELECT
    product_id,
    'Mar_sales' AS month,
    Mar_sales AS sales_amount
FROM monthly_sales
ORDER BY product_id, month;
```

save for later

Data Modeling and Table Relationships

Data modeling involves designing the structure of a database, including tables and their relationships. Common relationship types include one-to-one, one-to-many, and many-to-many.

Data Modeling and Table Relationships

follow for more

```sql
-- One-to-Many relationship example
CREATE TABLE departments (
    department_id INT PRIMARY KEY,
    department_name VARCHAR(50) NOT NULL
);

CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    employee_name VARCHAR(100) NOT NULL,
    department_id INT,
    FOREIGN KEY (department_id) REFERENCES
departments(department_id)
);

-- Many-to-Many relationship example
CREATE TABLE students (
    student_id INT PRIMARY KEY,
    student_name VARCHAR(100) NOT NULL
);

CREATE TABLE courses (
    course_id INT PRIMARY KEY,
    course_name VARCHAR(100) NOT NULL
);

CREATE TABLE enrollments (
    student_id INT,
    course_id INT,
    enrollment_date DATE,
    PRIMARY KEY (student_id, course_id),
    FOREIGN KEY (student_id) REFERENCES
students(student_id),
    FOREIGN KEY (course_id) REFERENCES
courses(course_id)
);
```

Swipe next →

save for later

Communicating Your Code

Clear communication of SQL code is crucial for collaboration and maintenance. Use comments, consistent formatting, and meaningful names for tables, columns, and aliases.

```sql
-- Calculate the average order value per
customer
-- for orders placed in the last 30 days
WITH recent_orders AS (
    SELECT
        customer_id,
        order_id,
        total_amount
    FROM orders
    WHERE order_date >= DATE_SUB(CURDATE(),
INTERVAL 30 DAY)
)
SELECT
    c.customer_name,
    COUNT(ro.order_id) AS order_count,
    AVG(ro.total_amount) AS avg_order_value
FROM customers c
LEFT JOIN recent_orders ro ON c.customer_id =
ro.customer_id
GROUP BY c.customer_id, c.customer_name
HAVING order_count > 0
ORDER BY avg_order_value DESC
LIMIT 10;
```

Swipe next

Turning Business Problems into Code

Translating business requirements into SQL involves understanding the problem, identifying relevant data, and breaking down the solution into logical steps.

Turning Business Problems into Code

save for later

```
-- Business Problem: Find top 5 products with
the highest revenue growth
-- compared to the same month last year
WITH monthly_revenue AS (
    SELECT
        p.product_id,
        p.product_name,
        EXTRACT(YEAR_MONTH FROM s.sale_date) AS
year_month,
        SUM(s.quantity * s.unit_price) AS
revenue
    FROM sales s
    JOIN products p ON s.product_id =
p.product_id
    WHERE s.sale_date >= DATE_SUB(CURDATE(),
INTERVAL 13 MONTH)
    GROUP BY p.product_id, p.product_name,
year_month
),
revenue_growth AS (
    SELECT
        cur.product_id,
        cur.product_name,
        cur.year_month,
        cur.revenue AS current_revenue,
        prev.revenue AS previous_revenue,
        (cur.revenue - prev.revenue) /
prev.revenue * 100 AS growth_percentage
    FROM monthly_revenue cur
    JOIN monthly_revenue prev ON
        cur.product_id = prev.product_id AND
        cur.year_month = prev.year_month + 100
    WHERE cur.year_month = EXTRACT(YEAR_MONTH
FROM CURDATE())
)
SELECT
    product_name,
    current_revenue,
    previous_revenue,
    growth_percentage
FROM revenue_growth
ORDER BY growth_percentage DESC
LIMIT 5;
```

Swipe next →

Query Optimization

Query optimization involves improving the performance of SQL queries. Techniques include proper indexing, avoiding subqueries when possible, and using EXPLAIN to analyze query execution plans.

save for later

Query Optimization

```sql
-- Before optimization
SELECT
    c.customer_name,
    COUNT(o.order_id) AS order_count
FROM customers c
LEFT JOIN orders o ON c.customer_id =
o.customer_id
WHERE o.order_date >= DATE_SUB(CURDATE(),
INTERVAL 1 YEAR)
GROUP BY c.customer_id, c.customer_name
HAVING order_count > 10
ORDER BY order_count DESC;

-- After optimization
SELECT
    c.customer_name,
    COUNT(o.order_id) AS order_count
FROM customers c
INNER JOIN (
    SELECT customer_id, order_id
    FROM orders
    WHERE order_date >= DATE_SUB(CURDATE(),
INTERVAL 1 YEAR)
) o ON c.customer_id = o.customer_id
GROUP BY c.customer_id, c.customer_name
HAVING order_count > 10
ORDER BY order_count DESC;

-- Add index to improve performance
CREATE INDEX idx_orders_customer_date ON orders
(customer_id, order_date);
```

Swipe next →

QAing Data

follow for more

Quality Assurance (QA) in SQL involves validating data integrity, consistency, and accuracy. This includes checking for null values, duplicate records, and ensuring data meets business rules.

```sql
-- Check for null values in important columns
SELECT
    COUNT(*) AS total_rows,
    COUNT(*) - COUNT(customer_id) AS
null_customer_id,
    COUNT(*) - COUNT(order_date) AS
null_order_date,
    COUNT(*) - COUNT(total_amount) AS
null_total_amount
FROM orders;

-- Identify duplicate orders
SELECT
    order_id,
    customer_id,
    order_date,
    COUNT(*) AS duplicate_count
FROM orders
GROUP BY order_id, customer_id, order_date
HAVING COUNT(*) > 1;

-- Ensure all products have a valid category
SELECT
    p.product_id,
    p.product_name,
    p.category_id
FROM products p
LEFT JOIN categories c ON p.category_id =
c.category_id
WHERE c.category_id IS NULL;
```

Swipe next →

save for later

Additional Resources

To further enhance your SQL skills, consider exploring the following resources:

1. "Efficient Query Processing for Data Science Workloads on Many-Core CPUs" by Orestis Polychroniou et al. (2019) ArXiv URL: https://arxiv.org/abs/1906.01560
2. "Automating the Database Schema Evolution Process" by Isak Karlsson et al. (2020) ArXiv URL: https://arxiv.org/abs/2010.05761
3. "Query Processing for Graph Analytics" by Angela Bonifati et al. (2020) ArXiv URL: https://arxiv.org/abs/2012.06889

Data scientist
&ML Engineer

Follow For More Data Science Content