SQL series

Advanced SQL Concepts

With Code Examples

Advanced Level

save for later

Advanced SQL Concepts

This presentation will cover several advanced SQL topics, including **Window Functions**, **Recursive Queries with CTEs, Advanced Joins and Subqueries, and more**.

These concepts are essential for working with complex data and solving intricate problems using SQL.

Swipe next →

follow for more

Window Functions

Window Functions are a powerful feature in SQL that allows you to perform calculations across a set of rows related to the current row. They provide a way to compute running totals, moving averages, rankings, and other analytical calculations.

```
CREATE TABLE sales_data (
    product_name VARCHAR(100),
    category VARCHAR(50),
    sales DECIMAL(10, 2)
);

INSERT INTO sales_data (product_name, category, sales)
VALUES
    ('Product A', 'Category 1', 1000.00),
    ('Product B', 'Category 1', 2000.00),
    ('Product C', 'Category 2', 1500.00),
    ('Product D', 'Category 2', 2500.00);
FROM
    sales_data;
```

Swipe next ⟶

save for later

Example

This query calculates the total sales for each category using the SUM window function with the PARTITION BY clause.

```
SELECT
    product_name,
    category,
    sales,
    SUM(sales) OVER (PARTITION BY category) AS category_total_sales
FROM
    sales_data;
```

Swipe next

follow for more

Recursive Queries with CTEs

```
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    employee_name VARCHAR(100),
    manager_id INT,
    FOREIGN KEY (manager_id) REFERENCES employees(employee_id)
);

INSERT INTO employees (employee_id, employee_name, manager_id)
VALUES
    (1, 'John Doe', NULL),
    (2, 'Jane Smith', 1),
    (3, 'Bob Johnson', 1),
    (4, 'Alice Williams', 2),
    (5, 'Tom Brown', 3);
```

Swipe next

save for later

Example

This recursive CTE traverses the employee hierarchy, starting from the top-level managers, and retrieves all employees with their respective levels in the hierarchy.

```
WITH RECURSIVE employee_hierarchy AS (
    SELECT
        employee_id,
        manager_id,
        employee_name,
        1 AS level
    FROM
        employees
    WHERE
        manager_id IS NULL
    UNION ALL
    SELECT
        e.employee_id,
        e.manager_id,
        e.employee_name,
        eh.level + 1
    FROM
        employees e
        INNER JOIN employee_hierarchy eh ON e.manager_id = eh.employee_id
)
SELECT
    *
FROM
    employee_hierarchy;
```

Swipe next

follow for more

Advanced Joins and Subqueries

```
CREATE TABLE products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(100),
    category_id INT,
    FOREIGN KEY (category_id) REFERENCES categories(category_id)
);

CREATE TABLE categories (
    category_id INT PRIMARY KEY,
    category_name VARCHAR(50)
);

CREATE TABLE sales_data (
    sale_id INT PRIMARY KEY,
    product_id INT,
    sales DECIMAL(10, 2),
    FOREIGN KEY (product_id) REFERENCES products(product_id)
);

INSERT INTO categories (category_id, category_name)
VALUES
    (1, 'Category 1'),
    (2, 'Category 2');

INSERT INTO products (product_id, product_name, category_id)
VALUES
    (1, 'Product A', 1),
    (2, 'Product B', 1),
    (3, 'Product C', 2),
    (4, 'Product D', 2);

INSERT INTO sales_data (sale_id, product_id, sales)
VALUES
    (1, 1, 1000.00),
    (2, 1, 500.00),
    (3, 2, 2000.00),
    (4, 3, 1500.00),
    (5, 4, 2500.00);
```

Swipe next

save for later

Example

This query combines data from the products and categories tables using an inner join and calculates the total sales for each product using a correlated subquery.

```
SELECT
    p.product_name,
    c.category_name,
    (
        SELECT
            SUM(sales)
        FROM
            sales_data sd
        WHERE
            sd.product_id = p.product_id
    ) AS total_sales
FROM
    products p
    INNER JOIN categories c ON p.category_id = c.category_id;
```

Swipe next

follow for more

Window Functions: Ranking Functions

This query ranks the products within each category based on their sales, using the RANK window function. (Use the sales_data table)

```
SELECT
    product_name,
    category,
    sales,
    RANK() OVER (PARTITION BY category ORDER BY sales DESC) AS rank_by_sales
FROM
    sales_data;
```

Swipe next

save for later

Recursive Queries: Generating Hierarchical Data

This recursive CTE generates a calendar table with dates from January 1, 2023, to December 31, 2023.

```
WITH RECURSIVE calendar AS (
    SELECT
        CAST('2023-01-01' AS DATE) AS date_value,
        1 AS level
    UNION ALL
    SELECT
        DATEADD(DAY, 1, date_value),
        level + 1
    FROM
        calendar
    WHERE
        date_value < '2023-12-31'
)
SELECT
    date_value
FROM
    calendar
ORDER BY
    date_value;
```

Swipe next ⟶

follow for more

Advanced Joins: Self-Joins

This query uses a self-join on the employees table to retrieve the employee names and their corresponding manager names. (Use the employees table from Slide 3.)

```sql
SELECT
    e1.employee_name AS employee,
    e2.employee_name AS manager
FROM
    employees e1
    INNER JOIN employees e2 ON e1.manager_id = e2.employee_id;
```

Swipe next

save for later

Subqueries in the FROM Clause

This query uses a subquery in the FROM clause to retrieve the product name, category, and sales data, and then calculates the total sales for each category. (Use the sales_data table)

```
SELECT
    category,
    SUM(sales) AS total_sales
FROM
    (
        SELECT
            product_name,
            category,
            sales
        FROM
            sales_data
    ) AS product_sales
GROUP BY
    category;
```

Swipe next

follow for more

Advanced Analytic Function

This query demonstrates the use of the **LEAD, LAG, FIRST_VALUE, LAST_VALUE, and NTH_VALUE,** which allow you to perform complex data analysis and calculations based on the ordering and partitioning of rows.

```
CREATE TABLE stock_prices (
    stock_symbol VARCHAR(10),
    trade_date DATE,
    open_price DECIMAL(10, 2),
    close_price DECIMAL(10, 2)
);

INSERT INTO stock_prices (stock_symbol, trade_date, open_price, close_price)
VALUES
    ('ACME', '2023-04-01', 100.00, 102.50),
    ('ACME', '2023-04-02', 103.00, 101.75),
    ('ACME', '2023-04-03', 102.25, 104.00),
    ('BXYZ', '2023-04-01', 50.00, 51.25),
    ('BXYZ', '2023-04-02', 51.50, 52.00),
    ('BXYZ', '2023-04-03', 51.75, 50.50);
```

Swipe next

save for later

Example

```sql
-- LAG example
SELECT
    stock_symbol,
    trade_date,
    open_price,
    close_price,
    LAG(open_price, 1) OVER (PARTITION BY stock_symbol ORDER BY trade_date)
AS previous_day_open
FROM
    stock_prices;

-- LEAD example (same as in the previous slide)
SELECT
    stock_symbol,
    trade_date,
    open_price,
    close_price,
    LEAD(close_price, 1) OVER (PARTITION BY stock_symbol ORDER BY
trade_date) AS next_day_close
FROM
    stock_prices;

-- FIRST_VALUE and LAST_VALUE examples
SELECT
    stock_symbol,
    trade_date,
    open_price,
    close_price,
    FIRST_VALUE(open_price) OVER (PARTITION BY stock_symbol ORDER BY
trade_date) AS first_open_price,
    LAST_VALUE(close_price) OVER (PARTITION BY stock_symbol ORDER BY
trade_date ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS
last_close_price
FROM
    stock_prices;

-- NTH_VALUE example
SELECT
    stock_symbol,
    trade_date,
    open_price,
    close_price,
    NTH_VALUE(close_price, 2) OVER (PARTITION BY stock_symbol ORDER BY
trade_date ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS
second_close_price
FROM
    stock_prices;
```

follow for more

Recursive Queries - Generating Sequences

Recursive queries can be used to generate sequences of numbers or dates, which can be useful for various purposes, such as generating test data or handling gaps in sequences.

```
WITH RECURSIVE number_sequence AS (
    SELECT 1 AS num
    UNION ALL
    SELECT num + 1
    FROM number_sequence
    WHERE num < 100
)
SELECT num
FROM number_sequence;
```

This recursive CTE generates a sequence of numbers from 1 to 100.

Swipe next ⟶

save for later

Correlated Subqueries

Correlated subqueries are subqueries that reference columns from the outer query. They can be used to perform complex filtering or calculations based on data from the outer query.

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    customer_id INT,
    order_date DATE,
    total_amount DECIMAL(10, 2)
);

INSERT INTO orders (order_id, customer_id, order_date, total_amount)
VALUES
    (1, 1, '2023-03-01', 100.00),
    (2, 1, '2023-03-15', 200.00),
    (3, 2, '2023-03-20', 150.00),
    (4, 2, '2023-04-01', 300.00);
```

Swipe next

follow for more

Example

While OOP offers many benefits, it's essential to be mindful of potential pitfalls, such as complexity due to deep inheritance hierarchies, tight coupling between classes, and misuse of design patterns. Striking the right balance and following best practices is crucial for maintainable and scalable code.

```sql
SELECT
    customer_id,
    order_id,
    order_date,
    total_amount,
    (
        SELECT MAX(total_amount)
        FROM orders o2
        WHERE o2.customer_id = o1.customer_id
            AND o2.order_date < o1.order_date
    ) AS previous_max_order
FROM
    orders o1;
```

save for later

Pitfalls and Best Practices

While advanced SQL concepts provide powerful capabilities, it's important to be aware of potential pitfalls and follow best practices:

- Optimize queries for performance, especially when dealing with large datasets or complex operations.
- Ensure data integrity and consistency when using recursive queries or hierarchical structures.
- Test thoroughly and validate results, especially when working with complex queries.
- Consider using database views or stored procedures for code organization and maintainability.
- Document your SQL code for better collaboration and future reference.