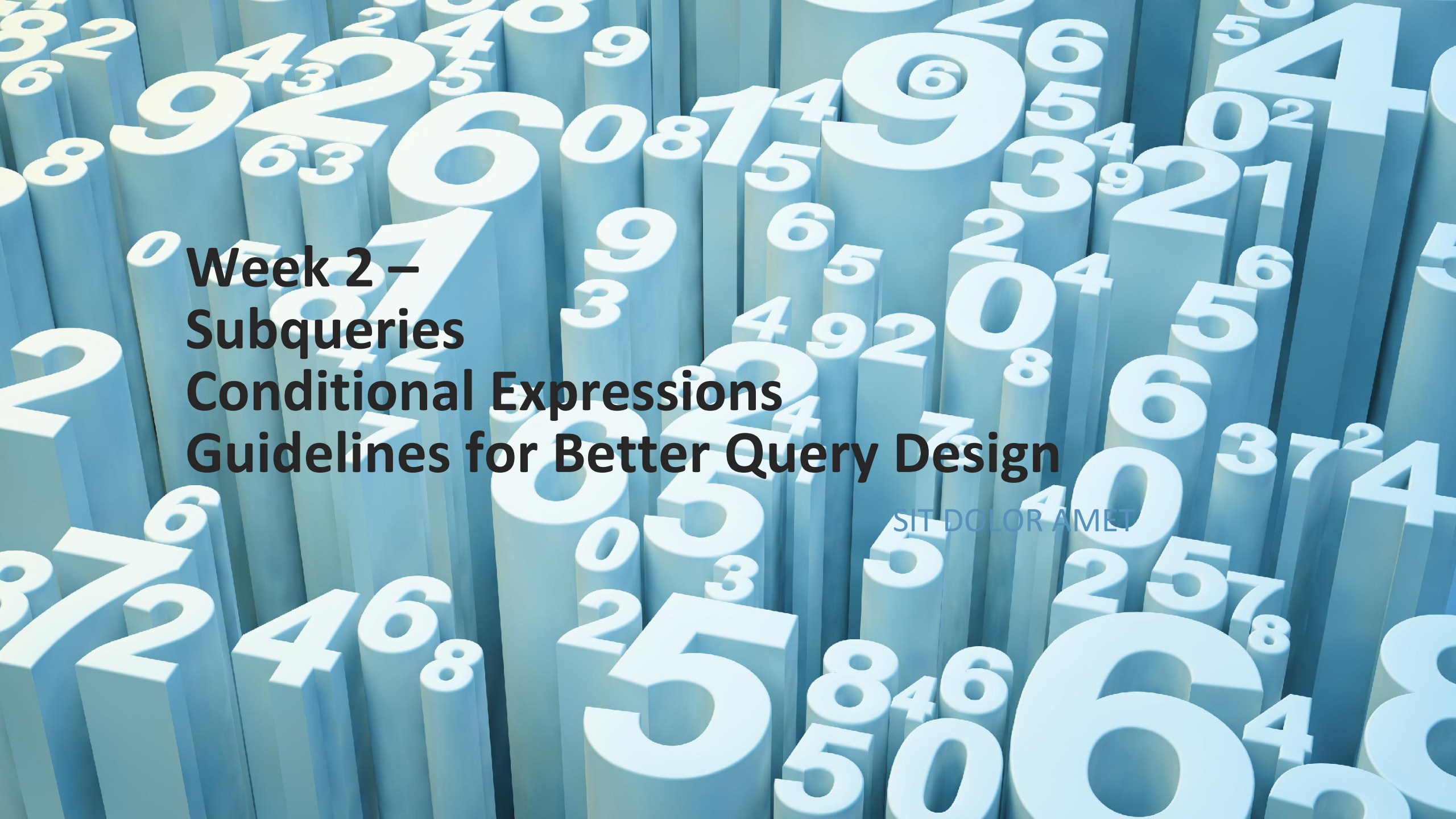

Week 2 – Subqueries Conditional Expressions Guidelines for Better Query Design

SIT DOLOR AMET

Subqueries

A subquery is a query nested inside another query. It provides a way to retrieve data that will be used in the main query.

Subqueries can be used in SELECT, INSERT, UPDATE, and DELETE statements.

Types of Subqueries:

- **Scalar Subquery**: Returns a single value.
- **Column Subquery**: Returns a single column with multiple rows.
- **Row Subquery**: Returns a single row with multiple columns.
- **Table Subquery**: Returns multiple rows and columns (a result set).

Examples

- **Scalar Subquery**: Used to find a single value to be compared.
- **Column Subquery**: Used to find a list of values.
- **Row Subquery**: Used to retrieve a single row of values.
- **Table Subquery**: Used for complex queries that return a table of results.

Scalar Subquery Example:

Scenario: Find the name of the employee with the highest salary.

Query:

SELECT employee_name

FROM employees

WHERE salary = (SELECT MAX(salary) FROM employees);

The subquery (SELECT MAX(salary) FROM employees) returns the highest salary.

The main query retrieves the name of the employee with this salary.

Column Subquery Example:

Scenario: Find employees who work in departments with more than 10 employees.

Query:

SELECT employee_name

FROM employees

WHERE department_id IN (SELECT department_id FROM employees GROUP BY department_id HAVING COUNT(*) > 10);

The subquery identifies departments with more than 10 employees.

The main query retrieves employees from these departments.

Row Subquery Example:

Scenario: Find all employees who have the same department and position as a specific employee.

Query:

SELECT employee_name

FROM employees

WHERE (department_id, position) = (SELECT department_id, position FROM employees WHERE employee_id = 101);

The subquery retrieves the department and position of a specific employee.

The main query finds all employees with the same department and position.

Table Subquery Example:

Scenario: Retrieve all departments with the average salary of employees.

Query:

SELECT department_id, AVG(salary) AS avg_salary

FROM employees

GROUP BY department_id;

The subquery here is used within a GROUP BY clause to calculate the average salary for each department.

Key Points and Best Practices

Subqueries in WHERE Clause: Useful for filtering results based on the results of another query.

Subqueries in FROM Clause: Used as a derived table to simplify complex queries.

Subqueries in SELECT Clause: Can be used to return computed values or aggregate results.

Best practices:

Performance: Ensure subqueries are optimized and consider indexing columns used in subqueries.

Readability: Use meaningful aliases and keep subqueries simple for clarity.

Avoid Nesting Too Deeply: Deeply nested subqueries can be harder to maintain and understand.

Conditional Expressions

- Conditional expressions allow for conditional logic within SQL queries.
- They help you transform and categorize data based on conditions.

Common Types

- CASE: The primary SQL conditional expression.
- IF and IIF: Available in some SQL dialects (e.g., MySQL, SQL Server).

Simple CASE Statement

A **CASE** statement evaluates a list of conditions and returns one of multiple possible result expressions.

Syntax:

```
SELECT column_name,
    CASE
        WHEN condition1 THEN result1
        WHEN condition2 THEN result2
        ELSE resultN
    END as alias_name
FROM table_name;
```

Example

```
SELECT product_name,
    CASE
        WHEN quantity > 50 THEN 'High Stock'
        WHEN quantity BETWEEN 10 AND 50 THEN 'Medium Stock'
        ELSE 'Low Stock'
    END as stock_level
FROM products;
```

Searched CASE Statement

A Searched CASE is similar to a simple CASE, but conditions can be complex expressions, not just equality checks.

Syntax:

```
SELECT column_name,
    CASE
        WHEN complex_condition1 THEN result1
        WHEN complex_condition2 THEN result2
        ELSE resultN
    END as alias_name
FROM table_name;
```

Example

```
SELECT employee_name,
    CASE
        WHEN salary > 100000 THEN 'High Salary'
        WHEN salary BETWEEN 50000 AND 100000 THEN 'Medium Salary'
        ELSE 'Low Salary'
    END as salary_level
FROM employees;
```

Using CASE in Aggregations

Example:

```
SELECT department_id,
    SUM(CASE WHEN salary > 100000 THEN 1 ELSE 0 END) as high_salary_count
FROM employees
GROUP BY department_id;
```

This query counts the number of employees with high salaries in each department.

Best Practices and Use Cases

- **Keep it Simple**: Avoid overly complex CASE statements that can make queries difficult to read and maintain.
- **Use for Data Transformation**: Conditional expressions are particularly useful for categorizing data, calculating derived values, and handling missing or exceptional data.
- **Example**: Transforming null values to a default value using CASE.

Guidelines for Better Query Design

- **Why Query Design Matters**:
  - Well-designed queries are more efficient, easier to maintain, and less prone to errors.
  - Poorly designed queries can lead to performance bottlenecks, especially in large databases.

Best Practices for Query Design

- **Select Only the Columns You Need**
  - Avoid using SELECT *. Instead, specify only the columns you need.
  - Example
    SELECT employee_name, salary FROM employees;
  - Reduces the amount of data processed and transferred, improving performance.

Use WHERE Clauses to Filter Data Early

- Apply filters as early as possible to reduce the number of rows processed.
  - Example:
    SELECT employee_name, salary FROM employees WHERE salary > 50000;
  - Helps the database engine focus on relevant data, speeding up query execution.

Use Joins Efficiently

- Join tables only when necessary, and choose the appropriate type of join (INNER JOIN, LEFT JOIN, etc.).
  - Example:
    SELECT e.employee_name, d.department_name
    FROM employees e
    INNER JOIN departments d ON e.department_id = d.department_id;
  - Minimizes the processing of irrelevant data and avoids unnecessary data duplication.

Use Indexes Appropriately

- Indexes can drastically improve query performance by allowing the database to locate data quickly.
  - Example:
    CREATE INDEX idx_employee_salary ON employees(salary);
  - Helps the database engine quickly locate the rows that match your query criteria.

Avoid Redundant Subqueries

- While subqueries are powerful, avoid using them when a JOIN or direct condition will suffice.
  - Example:
    SELECT product_name FROM products
    WHERE category_id = (SELECT category_id FROM categories WHERE category_name = 'Electronics');

    Improved with JOIN:
    SELECT p.product_name
    FROM products p
    INNER JOIN categories c ON p.category_id = c.category_id
    WHERE c.category_name = 'Electronics';
  - Using JOIN instead of a subquery can make the query more readable and often more efficient.