SQL - Advance

Reading Material

PW SKILLS

Window Functions

Window functions are advanced SQL features that allow you to perform calculations across a set of table rows that are somehow related to the current row. Unlike aggregate functions, which return a single value after aggregating data over a set of rows, window functions can return multiple rows for a set of data. This makes them particularly powerful for tasks like running totals, ranking, and moving averages.

- **Overview of window functions**

A window function operates over a "window" of rows that are defined by the OVER() clause. The window of rows can be defined by specifying:

1. **PARTITION BY:** Divides the result set into partitions to which the function is applied.
2. **ORDER BY:** Specifies the order of rows within each partition.
3. **Window frame:** Further restricts the set of rows within the partition.

Here's a general syntax for a window function:

```sql
window_function() OVER (
    [PARTITION BY partition_expression]
    [ORDER BY order_expression]
    [frame_clause]
)
```

- **ROW_NUMBER, RANK, DENSE_RANK, PARTITION BY**

These functions are categorized as **Window Functions** in SQL, which operate on a set of rows related to the current row. They are particularly useful for ranking, numbering, and partitioning result sets.

1. **ROW_NUMBER():** Assigns a unique sequential number to each row within a result set based on the order specified in the ORDER BY clause.
   - Syntax:

```
SQL

ROW_NUMBER() OVER (ORDER BY column_name ASC|DESC)

Use code with caution.
```

2. **RANK():** Assigns a rank to each row based on the order specified in the ORDER BY clause. If there are ties, the same rank is assigned to all tied rows, and the next rank is skipped.
   - Syntax:

```
SQL

RANK() OVER (ORDER BY column_name ASC|DESC)

Use code with caution.
```

3. **DENSE_RANK():** Similar to RANK(), but it assigns a rank to each row without gaps, even if there are ties.

- Syntax:

```
SQL

DENSE_RANK() OVER (ORDER BY column_name ASC|DESC)
```

Use code with caution.

4. **PARTITION BY:** Divides the result set into partitions based on specified columns. The ranking functions are then applied within each partition.

- Syntax:

```
SQL

ROW_NUMBER() OVER (PARTITION BY column_name ORDER BY column_name ASC|
```

Use code with caution.

Example: Ranking Employees by Salary
Sample Dataset:

```sql
CREATE TABLE employees (
    id INT,
    department VARCHAR(50),
    name VARCHAR(50),
    salary DECIMAL(10, 2)
);

INSERT INTO employees (id, department, name, salary) VALUES
(1, 'HR', 'Alice', 60000),
(2, 'HR', 'Bob', 75000),
(3, 'HR', 'Charlie', 75000),
(4, 'IT', 'David', 80000),
(5, 'IT', 'Eve', 90000),
(6, 'IT', 'Frank', 85000);
```

Query:

```sql
SELECT
    department,
    name,
    salary,
    ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) AS row_number,
    RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS rank,
    DENSE_RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS dense_rank
FROM
    employees;
```

Output:

department	name	salary	row_number	rank	dense_rank
HR	Bob	75000.00	1	1	1
HR	Charlie	75000.00	2	1	1
HR	Alice	60000.00	3	3	2
IT	Eve	90000.00	1	1	1
IT	Frank	85000.00	2	2	2
IT	David	80000.00	3	3	3

- ROW_NUMBER(): Each row is assigned a unique number starting from 1 within each partition.
- RANK(): The ranking is the same for ties, but gaps are left in the rank sequence.
- DENSE_RANK(): The ranking is the same for ties, with no gaps in the rank sequence.
- PARTITION BY: Divides the result set into partitions based on Department.

- **LAG, LEAD, FIRST_VALUE, LAST_VALUE**

These window functions are used to access data from other rows relative to the current row within the same partition.

LAG(column, offset, default): Provides access to a value from a row at a specified physical offset before the current row.

LEAD(column, offset, default): Provides access to a value from a row at a specified physical offset after the current row.

FIRST_VALUE(column): Returns the first value in the ordered set of a partition.

LAST_VALUE(column): Returns the last value in the ordered set of a partition.

Example: Analyzing Salary Trends within Departments

Query to Analyze Salary Trends:

sql Copy code

```sql
SELECT
    department,
    name,
    salary,
    LAG(salary, 1, 0) OVER (PARTITION BY department ORDER BY salary DESC) AS prev_salary,
    LEAD(salary, 1, 0) OVER (PARTITION BY department ORDER BY salary DESC) AS next_salary,
    FIRST_VALUE(salary) OVER (PARTITION BY department ORDER BY salary DESC) AS first_salary,
    LAST_VALUE(salary) OVER (PARTITION BY department ORDER BY salary DESC
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS last_salary
FROM
    employees;
```

LAG(): Retrieves the salary from the previous row in the same partition. If there is no previous row, it returns the default value (0 in this case).

LEAD(): Retrieves the salary from the next row in the same partition. If there is no next row, it returns the default value (0).

FIRST_VALUE(): Returns the highest salary within each department (as determined by the ORDER BY clause).

LAST_VALUE(): Returns the lowest salary within each department. The frame clause ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING ensures it considers the entire partition.

Output:

department	name	salary	prev_salary	next_salary	first_salary	last_salary
HR	Bob	75000.00	0	75000.00	75000.00	60000.00
HR	Charlie	75000.00	75000.00	60000.00	75000.00	60000.00
HR	Alice	60000.00	75000.00	0	75000.00	60000.00
IT	Eve	90000.00	0	85000.00	90000.00	80000.00
IT	Frank	85000.00	90000.00	80000.00	90000.00	80000.00
IT	David	80000.00	85000.00	0	90000.00	80000.00

LAG(): Helps compare each employee's salary with the previous one.

LEAD(): Helps compare each employee's salary with the next one.

FIRST_VALUE(): Identifies the top salary within each department.

LAST_VALUE(): Identifies the lowest salary within each department.

- **Window frame specification**

The window frame clause allows you to define a subset of rows relative to the current row within a partition. This is particularly useful for calculations like running totals, moving averages, and cumulative distributions.

Window Frame Clause Syntax:

```sql
ROWS|RANGE BETWEEN <start> AND <end>
```

Where:

- **ROWS:** Refers to a specific number of rows relative to the current row.
- **RANGE:** Refers to a range of values relative to the current row.

Common Frame Specifications:

ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW: All rows from the start of the partition to the current row.

ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING: The current row and all rows that follow in the partition.

ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING: One row before, the current row, and one row after.

Example: Calculating Running Totals

Query to Calculate Running Total of Salaries:

```sql
SELECT
    department,
    name,
    salary,
    SUM(salary) OVER (PARTITION BY department ORDER BY salary DESC
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS running_total
FROM
    employees;
```

This query calculates a running total of salaries within each department, starting from the highest salary down to the current row. The window frame **ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW** ensures that the sum includes all preceding rows in the partition.

Output:

department	name	salary	running_total
HR	Bob	75000.00	75000.00
HR	Charlie	75000.00	150000.00
HR	Alice	60000.00	210000.00
IT	Eve	90000.00	90000.00
IT	Frank	85000.00	175000.00
IT	David	80000.00	255000.00

ו. This

Running Total: The SUM() function aggregates the salaries cumulatively for each department.

Case Statement and Temporary Tables

SQL provides powerful tools to handle complex data manipulations and conditional logic through CASE statements and temporary tables. These features are indispensable when dealing with dynamic queries, conditional aggregations, or when intermediate result storage is required. This explanation will delve into the detailed concepts of CASE WHEN expressions and temporary tables, including their syntax, usage scenarios, and practical examples with sample data.

- **CASE WHEN expressions**

The CASE statement in SQL is a conditional expression that evaluates a list of conditions and returns one of multiple possible result expressions. It's similar to IF-THEN-ELSE logic found in programming languages. The CASE statement can be used in SELECT, UPDATE, DELETE, and even in ORDER BY clauses. It is particularly useful for performing complex logic within a single SQL query without the need for additional joins or subqueries.

Syntax

```sql
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    ...
    ELSE default_result
END
```

condition1, condition2, ...: These are the conditions that are checked sequentially. If a condition is true, the corresponding result is returned.
result1, result2, ...: These are the results returned when the corresponding condition is true.
default_result: This is the result returned if none of the conditions are met. This part is optional.

Usage Scenarios

- **Conditional Data Transformation:** Modify data on the fly based on specific conditions.
- **Dynamic Grouping or Categorization:** Group or categorize data differently depending on specific criteria.
- **Custom Aggregations:** Perform aggregations that depend on specific conditions.

Example 1: Simple CASE WHEN Expression

Scenario: Categorize employees based on their salary into three categories: Low, Medium, and High.

Sample Data:

Employee_ID	Employee_Name	Salary
1	Alice	60,000
2	Bob	75,000
3	Charlie	50,000
4	David	90,000
5	Eve	40,000

Query:

```sql
SELECT
    Employee_Name,
    Salary,
    CASE
        WHEN Salary < 50000 THEN 'Low'
        WHEN Salary BETWEEN 50000 AND 80000 THEN 'Medium'
        ELSE 'High'
    END AS Salary_Category
FROM
    Employees;
```

Output:

Employee_Name	Salary	Salary_Category
Alice	60,000	Medium
Bob	75,000	Medium
Charlie	50,000	Medium
David	90,000	High
Eve	40,000	Low

The query categorizes each employee's salary into 'Low', 'Medium', or 'High' based on the conditions provided in the CASE statement.

Example 2: Using CASE in an Aggregate Function

Scenario: Calculate the total salaries of employees categorized by salary levels (Low, Medium, High).

Query:

```sql
SELECT
    CASE
        WHEN Salary < 50000 THEN 'Low'
        WHEN Salary BETWEEN 50000 AND 80000 THEN 'Medium'
        ELSE 'High'
    END AS Salary_Category,
    SUM(Salary) AS Total_Salary
FROM
    Employees
GROUP BY
    CASE
        WHEN Salary < 50000 THEN 'Low'
        WHEN Salary BETWEEN 50000 AND 80000 THEN 'Medium'
        ELSE 'High'
    END;
```

Output:

Salary_Category	Total_Salary
Low	40,000
Medium	185,000
High	90,000

The query calculates the total salary for each category ('Low', 'Medium', 'High') using a CASE statement within the GROUP BY clause.

- **Temporary tables and their usage**

Temporary tables in SQL are used to store intermediate results that you might need to reference multiple times within the same session. These tables are particularly useful when working with complex queries, where breaking down the query into steps can simplify the logic or improve performance. Temporary tables are session-specific, meaning they are only accessible within the session in which they were created and are automatically dropped when the session ends.

Syntax

Creating a temporary table can be done using the `CREATE TEMPORARY TABLE` statement:

```sql
CREATE TEMPORARY TABLE temp_table_name AS
SELECT column1, column2, ...
FROM existing_table
WHERE conditions;
```

- **temp_table_name**: The name of the temporary table.
- **AS SELECT ...**: This clause defines the structure and content of the temporary table, typically copied from an existing query.

Usage Scenarios

Simplifying Complex Queries: Break down a complex query into simpler parts by storing intermediate results.

Performance Optimization: Reduce the number of times a complex query needs to be run by storing the result in a temporary table.

Data Transformation: Perform multiple transformations on a dataset before producing the final result.

Example: Creating and Using Temporary Tables

Scenario: Suppose you want to analyze the employees who have salaries above a certain threshold and then categorize these employees.

Sample Data:

Employee_ID	Employee_Name	Salary
1	Alice	60,000
2	Bob	75,000
3	Charlie	50,000
4	David	90,000
5	Eve	40,000

Step 1: Create a Temporary Table to Store Employees with Salary Above 55,000

```sql
CREATE TEMPORARY TABLE High_Earners AS
SELECT
    Employee_ID,
    Employee_Name,
    Salary
FROM
    Employees
WHERE
    Salary > 55000;
```

This query creates a temporary table named **High_Earners** that stores only those employees whose salary is greater than 55,000.

Step 2: Use the Temporary Table to Categorize High Earners

```sql
SELECT
    Employee_Name,
    Salary,
    CASE
        WHEN Salary BETWEEN 55001 AND 75000 THEN 'Medium'
        ELSE 'High'
    END AS Salary_Category
FROM
    High_Earners;
```

Output:

Employee_Name	Salary	Salary_Category
Alice	60,000	Medium
Bob	75,000	Medium
David	90,000	High

This query uses the **High_Earners** temporary table to categorize the salaries of employees into 'Medium' and 'High' categories.

Stored Procedures

Stored procedures are a powerful feature in SQL that allows you to encapsulate a sequence of SQL statements into a reusable unit. They are essentially precompiled collections of SQL statements that can be executed as a single command. Stored procedures help in improving performance, enhancing security, and promoting code reuse.

- **Creating and executing stored procedures**

A stored procedure is a group of SQL statements that can be executed on the database server. It is stored in the database and can be invoked with a simple call. Stored procedures can accept input parameters, return output parameters, and even perform complex operations like loops and conditionals.

Syntax for Creating a Stored Procedure:

```sql
CREATE PROCEDURE procedure_name
AS
BEGIN
    -- SQL statements
END;
```

Example: Basic Stored Procedure

Let's consider a scenario where we want to create a stored procedure to retrieve all employees with a salary greater than a certain amount.

Sample Data:

Employee_ID	Employee_Name	Salary
1	Alice	60,000
2	Bob	75,000
3	Charlie	50,000
4	David	90,000
5	Eve	40,000

Creating the Stored Procedure:

```sql
CREATE PROCEDURE GetHighSalaryEmployees
    @MinSalary DECIMAL
AS
BEGIN
    SELECT Employee_ID, Employee_Name, Salary
    FROM Employees
    WHERE Salary > @MinSalary;
END;
```

Executing the Stored Procedure:
To execute the above-stored procedure and retrieve employees with salaries greater than $60,000, you would run:

```sql
EXEC GetHighSalaryEmployees @MinSalary = 60000;
```

Output:

Employee_ID	Employee_Name	Salary
2	Bob	75,000
4	David	90,000

- **Input and output parameters**

Parameters allow stored procedures to be more flexible by passing values at runtime. Input parameters are used to send values into a stored procedure, while output parameters are used to return values from the stored procedure to the caller.

Input Parameters:

An input parameter is defined by specifying a parameter name and its data type. It allows you to pass values into the stored procedure.

Example:
The GetHighSalaryEmployees procedure shown above uses an input parameter @MinSalary to filter employees based on their salary.

Output Parameters:
Output parameters are used when you want the stored procedure to return a value. They are defined using the OUTPUT keyword.

Example: Stored Procedure with Output Parameter
Suppose you want to create a stored procedure that calculates the average salary of all employees and returns it.

```sql
CREATE PROCEDURE GetAverageSalary
    @AvgSalary DECIMAL OUTPUT
AS
BEGIN
    SELECT @AvgSalary = AVG(Salary)
    FROM Employees;
END;
```

Executing the Stored Procedure with Output Parameter:

```sql
DECLARE @AverageSalary DECIMAL;
EXEC GetAverageSalary @AvgSalary = @AverageSalary OUTPUT;
SELECT @AverageSalary AS AverageSalary;
```

Output:

AverageSalary
63,000

- **Advantages of stored procedures**

Stored procedures offer several benefits that can significantly enhance the efficiency, security, and maintainability of database operations.

Advantages:

1. **Performance Improvement:**
Stored procedures are precompiled and stored in the database, which means they execute faster than individual SQL queries sent from the application layer. The database engine can cache execution plans, leading to faster query execution.

2. **Reduced Network Traffic:**
Instead of sending multiple SQL queries over the network, you can send a single stored procedure call, reducing the amount of data transferred between the application and the database server.

3. **Enhanced Security:**
Stored procedures can encapsulate business logic and restrict direct access to the underlying tables. Permissions can be granted to execute the procedure without granting access to the tables themselves, thus providing an additional security layer.

4. **Code Reusability:**
Commonly used SQL statements can be encapsulated within stored procedures, promoting code reuse and reducing duplication. This makes maintaining the code easier, as changes need to be made only in one place.

5. **Easier Maintenance:**
Centralizing business logic in stored procedures makes it easier to maintain and update the application. When business rules change, you only need to modify the stored procedure, without changing the application code.

6. **Support for Complex Operations:**
Stored procedures can handle complex logic, including conditional statements (IF...ELSE), loops (WHILE), and error handling (TRY...CATCH), making them suitable for implementing advanced business logic.

Example of Enhanced Security:
Consider a scenario where only authorized users should be able to update employee salaries. By using a stored procedure, you can enforce this security rule.

sql Copy code

```sql
CREATE PROCEDURE UpdateEmployeeSalary
    @EmployeeID INT,
    @NewSalary DECIMAL
AS
BEGIN
    UPDATE Employees
    SET Salary = @NewSalary
    WHERE Employee_ID = @EmployeeID;
END;
```

Here, you can grant execution rights to this procedure without granting direct access to the Employees table, thereby securing the data.

Functions:

Functions in SQL are a powerful feature that allows you to encapsulate a set of SQL operations or logic into a single unit that can be called and reused multiple times. They are particularly useful in situations where the same logic or operation is required in multiple queries or reports. Functions can accept parameters, perform calculations or manipulations, and return a value. In SQL, functions can be broadly categorized into three main types: User-Defined Functions (UDFs), Scalar Functions, Table-Valued Functions, and Aggregate Functions.

- **User-defined functions (UDFs)**

User-Defined Functions (UDFs) are custom functions created by the user to perform operations that are not built into SQL by default. These functions allow users to define their own logic, which can be reused across multiple queries. UDFs can accept parameters, perform complex calculations or operations, and return a result.

Types of UDFs:

Scalar Functions: Return a single value.
Table-Valued Functions (TVFs): Return a table.
Example of a Scalar UDF:
Suppose we have a scenario where we need to calculate the total price of an item after applying a discount. We can create a scalar UDF to perform this calculation.

Sample Data:

Item_ID	Item_Name	Price	Discount_Percent
1	Laptop	1000	10
2	Mouse	50	5
3	Keyboard	75	15

UDF Definition:

```sql
CREATE FUNCTION CalculateDiscountedPrice (@Price DECIMAL, @Discount DECIMAL)
RETURNS DECIMAL
AS
BEGIN
    RETURN @Price - (@Price * @Discount / 100);
END;
```

Usage Example:

```sql
SELECT Item_Name, Price, Discount_Percent,
       dbo.CalculateDiscountedPrice(Price, Discount_Percent) AS Discounted_Price
FROM Items;
```

Output:

Item_Name	Price	Discount_Percent	Discounted_Price
Laptop	1000	10	900
Mouse	50	5	47.5
Keyboard	75	15	63.75

- **Scalar functions, table-valued functions**

Scalar functions
Scalar functions are a type of UDF that return a single value, such as a number, string, or date. They are often used to perform operations like formatting strings, performing calculations, or manipulating dates.

Example of a Scalar Function:
Let's say we want to create a function that converts a full name into an email-friendly format by removing spaces and converting the name to lowercase.

Scalar Function Definition:

```sql
CREATE FUNCTION FormatEmail (@FullName NVARCHAR(100))
RETURNS NVARCHAR(100)
AS
BEGIN
    RETURN LOWER(REPLACE(@FullName, ' ', '.')) + '@example.com';
END;
```

Usage Example:

```sql
SELECT Employee_Name, dbo.FormatEmail(Employee_Name) AS Email
FROM Employees;
```

Sample Data:

Employee_Name
John Doe
Jane Smith
Bob Johnson

Output:

Employee_Name	Email
John Doe	john.doe@example.com
Jane Smith	jane.smith@example.com
Bob Johnson	bob.johnson@example.com

Table-valued functions

Table-Valued Functions (TVFs) return a table as a result, allowing you to treat the result set as if it were a regular table. TVFs are particularly useful when you need to encapsulate complex queries that return multiple rows and columns.

Example of a Table-Valued Function:

Let's create a TVF that returns all employees in a specific department, given the department ID as a parameter.

Sample Data:

Employee_ID	Employee_Name	Department_ID
1	John Doe	10
2	Jane Smith	20
3	Bob Johnson	10

TVF Definition:

```sql
CREATE FUNCTION GetEmployeesByDepartment (@DeptID INT)
RETURNS TABLE
AS
RETURN
(
    SELECT Employee_ID, Employee_Name
    FROM Employees
    WHERE Department_ID = @DeptID
);
```

Usage Example:

```sql
SELECT *
FROM dbo.GetEmployeesByDepartment(10);
```

Output:

Employee_ID	Employee_Name
1	John Doe
3	Bob Johnson

- **Aggregate functions**

Aggregate functions are built-in SQL functions that perform a calculation on a set of values and return a single value. Common aggregate functions include SUM, AVG, COUNT, MIN, and MAX.

Example of Aggregate Functions:

Consider a table of sales transactions. We want to calculate the total sales, average sales amount, and the number of transactions.

Sample Data:

Transaction_ID	Sale_Amount
1	150
2	200
3	100

Aggregate Functions Usage:

```sql
SELECT
    SUM(Sale_Amount) AS Total_Sales,
    AVG(Sale_Amount) AS Average_Sales,
    COUNT(Transaction_ID) AS Number_of_Transactions
FROM Sales;
```

Output:

Total_Sales	Average_Sales	Number_of_Transactions
450	150	3

Views and Cursor

- **Creating and using views**

A view in SQL is a virtual table that is created by querying one or more tables. Unlike a regular table, a view does not store data physically. Instead, it dynamically retrieves data from the underlying tables whenever it is accessed. Views provide a way to simplify complex queries, present data in a specific format, or restrict access to sensitive data. They act as a layer of abstraction between the user and the actual tables, making it easier to manage and manipulate data.

Creating and Using Views:

A view is created using the CREATE VIEW statement, followed by a query that defines the content of the view. Once created, a view can be queried just like a regular table.

Syntax:

```sql
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Example:

Consider a database for an online store with a table named Orders that stores information about customer orders.

Sample Data:

Order_ID	Customer_Name	Product_Name	Quantity	Price	Order_Date
1	Alice Brown	Laptop	1	1200.00	2024-08-01
2	Bob Smith	Tablet	2	600.00	2024-08-03
3	Carol Jones	Smartphone	3	300.00	2024-08-05
4	David White	Laptop	1	1300.00	2024-08-06
5	Emma Taylor	Tablet	1	650.00	2024-08-07

Let's create a view that displays only the orders for laptops:

```sql
CREATE VIEW Laptop_Orders AS
SELECT Order_ID, Customer_Name, Product_Name, Quantity, Price
FROM Orders
WHERE Product_Name = 'Laptop';
```

Output of View:

When you query the view `Laptop_Orders`, it will return the following result:

```sql
SELECT * FROM Laptop_Orders;
```

Order_ID	Customer_Name	Product_Name	Quantity	Price
1	Alice Brown	Laptop	1	1200.00
4	David White	Laptop	1	1300.00

Advantages of Using Views:

1. **Simplification:** Views can simplify complex queries by encapsulating them into a single query that can be reused.
2. **Security:** Views can restrict access to specific columns or rows, providing an additional layer of security.
3. **Data Abstraction:** Views provide a way to present data in a format that is more meaningful to the user.
4. **Data Independence:** Views can help insulate users from changes in the underlying table structures.

Updating Data Through Views:

In some cases, views can be used to update the underlying data. However, there are restrictions, especially if the view is based on complex queries involving joins, aggregations, or calculated columns.

Example:

```sql
UPDATE Laptop_Orders
SET Price = 1250.00
WHERE Order_ID = 1;
```

This query updates the price of the laptop in the Orders table through the Laptop_Orders view.

- **Cursors for row-by-row processing**

A cursor in SQL is a database object used to retrieve, manipulate, and process rows returned by a query, one at a time. Unlike a typical SQL query that operates on an entire result set at once, a cursor allows for row-by-row processing, which is particularly useful when you need to perform complex logic on each row of a result set. Cursors are often used in situations where operations cannot be efficiently performed using standard SQL queries alone, such as iterating over a result set to perform row-specific calculations, validations, or updates.

Creating and Using Cursors:

The process of working with a cursor involves several steps:

1. **Declare the Cursor:** Define the cursor with the query that determines the result set.
2. **Open the Cursor:** Execute the query and establish the result set.
3. **Fetch Data:** Retrieve each row from the result set, one at a time.
4. **Process Data:** Perform operations on each row as needed.
5. **Close the Cursor:** Release the cursor resources after processing is complete.
6. **Deallocate the Cursor:** Free the resources associated with the cursor.

Syntax:

sql Copy code

```sql
DECLARE cursor_name CURSOR FOR
SELECT column1, column2, ...
FROM table_name
WHERE condition;

OPEN cursor_name;

FETCH NEXT FROM cursor_name INTO variable1, variable2, ...;

-- Process each row
WHILE @@FETCH_STATUS = 0
BEGIN
    -- Your logic here
    FETCH NEXT FROM cursor_name INTO variable1, variable2, ...;
END;

CLOSE cursor_name;
DEALLOCATE cursor_name;
```

Example:
Let's consider a scenario where we want to iterate over the Orders table to calculate a discount for each order based on the total order value.

Sample Data:

Order_ID	Customer_Name	Product_Name	Quantity	Price
1	Alice Brown	Laptop	1	1200.00
2	Bob Smith	Tablet	2	600.00
3	Carol Jones	Smartphone	3	300.00
4	David White	Laptop	1	1300.00
5	Emma Taylor	Tablet	1	650.00

We want to give a 10% discount to orders where the total price is greater than $1000.

Cursor Implementation:

sql Copy code

```sql
DECLARE @OrderID INT, @TotalPrice DECIMAL(10, 2);

DECLARE OrderCursor CURSOR FOR
SELECT Order_ID, Quantity * Price AS TotalPrice
FROM Orders;

OPEN OrderCursor;

FETCH NEXT FROM OrderCursor INTO @OrderID, @TotalPrice;

WHILE @@FETCH_STATUS = 0
BEGIN
    IF @TotalPrice > 1000
    BEGIN
        UPDATE Orders
        SET Price = Price * 0.9
        WHERE Order_ID = @OrderID;
    END

    FETCH NEXT FROM OrderCursor INTO @OrderID, @TotalPrice;
END;

CLOSE OrderCursor;
DEALLOCATE OrderCursor;
```

Output:

After executing the above cursor, the `Orders` table will have updated prices for orders where the total value exceeded $1000.

Order_ID	Customer_Name	Product_Name	Quantity	Price
1	Alice Brown	Laptop	1	1080.00
2	Bob Smith	Tablet	2	600.00
3	Carol Jones	Smartphone	3	300.00
4	David White	Laptop	1	1170.00
5	Emma Taylor	Tablet	1	650.00

Advantages of Using Cursors:

1. **Row-by-Row Processing:** Cursors allow for fine-grained control over individual rows, making them ideal for tasks that require iterative logic.
2. **Complex Operations:** Cursors are useful for performing complex operations that cannot be easily expressed in a single SQL query.
3. **Sequential Access:** Cursors provide a way to process result sets sequentially, which can be necessary in certain scenarios.

Disadvantages of Using Cursors:

1. **Performance Overhead:** Cursors can be slower than set-based operations because they process rows one at a time.
2. **Resource-Intensive:** Cursors consume more database resources (memory, CPU) compared to regular queries.
3. **Complexity:** Writing and managing cursor-based logic can be more complex and error-prone than using set-based SQL.

Triggers

Triggers are a powerful feature in SQL that allows you to automatically execute a batch of SQL statements in response to specific events on a table or view. Triggers are essential for maintaining data integrity, enforcing business rules, and automating tasks in a database.

- **Creating and managing triggers**

A trigger is a special type of stored procedure that automatically runs when an event occurs in the database server. The events that can activate a trigger include data modifications like INSERT, UPDATE, or DELETE operations. Triggers are defined to respond either before or after the triggering event.

Triggers can be used to:

1. Enforce complex business rules.
2. Automatically audit changes to data.
3. Synchronize tables.
4. Prevent unauthorized changes.

Syntax:

Here's the basic syntax for creating a trigger:

sql Copy code

```sql
CREATE TRIGGER trigger_name
{BEFORE | AFTER} {INSERT | UPDATE | DELETE}
ON table_name
FOR EACH ROW
BEGIN
    -- Trigger logic goes here
END;
```

1. trigger_name: **The name of the trigger.**
2. BEFORE | AFTER: **Specifies whether the trigger should be executed before or after the triggering event.**
3. INSERT | UPDATE | DELETE: **The event that will activate the trigger.**
4. table_name:**The table associated with the trigger.**
5. FOR EACH ROW:**The trigger will fire for each row affected by the triggering event.**

Sample Data:

Employee_ID	Name	Salary
1	John Doe	50000
2	Jane Smith	55000
3	Mark Jones	60000

Trigger Creation:

```sql
CREATE TRIGGER LogSalaryUpdate
AFTER UPDATE ON Employees
FOR EACH ROW
BEGIN
   INSERT INTO SalaryLog (Employee_ID, Old_Salary, New_Salary, Change_Date)
   VALUES (OLD.Employee_ID, OLD.Salary, NEW.Salary, NOW());
END;
```

1. Trigger Name: LogSalaryUpdate
2. AFTER UPDATE: The trigger will execute after an UPDATE operation on the Employees table.
3. FOR EACH ROW: It applies to every row affected by the update.

Trigger Logic: The trigger inserts a record into a SalaryLog table, capturing the Employee_ID, old salary, new salary, and the date of change.

Output:
If John Doe's salary is updated from $50,000 to $52,000, the SalaryLog table will have a new entry like:

Log_ID	Employee_ID	Old_Salary	New_Salary	Change_Date
1	1	50000	52000	2024-08-14 10:00:00

- **After and before triggers**

Triggers can be defined to run either before or after the triggering event.

Before Triggers: These triggers are executed before the actual data modification (insert, update, delete) occurs. They are often used for validation or modification of input data before it is committed to the database.

After Triggers: These triggers are executed after the data modification. They are commonly used for actions like logging changes, updating related tables, or enforcing referential integrity.

Example of a Before Trigger:
Let's create a trigger that ensures no employee can have a salary lower than $30,000 before inserting or updating a record.

```sql
CREATE TRIGGER CheckSalary
BEFORE INSERT OR UPDATE ON Employees
FOR EACH ROW
BEGIN
   IF NEW.Salary < 30000 THEN
      SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'Salary cannot be less than 30000';
   END IF;
END;
```

1. **Trigger Name:** CheckSalary
2. **BEFORE INSERT OR UPDATE:** The trigger fires before an INSERT or UPDATE operation.
3. **Trigger Logic:** The trigger checks if the new salary (NEW.Salary) is below $30,000. If it is, an error is raised, and the operation is aborted.

Example of an After Trigger:

Let's create an after trigger that updates a department's total salary expense whenever an employee's salary is updated.

```sql
CREATE TRIGGER UpdateDeptSalary
AFTER UPDATE ON Employees
FOR EACH ROW
BEGIN
   UPDATE Departments
   SET Total_Salary = Total_Salary - OLD.Salary + NEW.Salary
   WHERE Department_ID = NEW.Department_ID;
END;
```

1. **Trigger Name:** UpdateDeptSalary
2. **AFTER UPDATE:** The trigger fires after an UPDATE operation on the Employees table.
3. **Trigger Logic:** The trigger adjusts the Total_Salary of the associated department by subtracting the old salary and adding the new salary.

- **Insert, update, and delete triggers**

An INSERT trigger is executed when a new row is inserted into the table. This can be used to automatically fill certain fields, enforce rules, or log data.

Example:
Let's create a trigger that automatically adds an entry to an audit table whenever a new employee is added.

```sql
CREATE TRIGGER InsertEmployeeAudit
AFTER INSERT ON Employees
FOR EACH ROW
BEGIN
   INSERT INTO AuditLog (Action, Employee_ID, Change_Date)
   VALUES ('INSERT', NEW.Employee_ID, NOW());
END;
```

Update Triggers:

An UPDATE trigger is fired when a row is updated in the table. It can be used to track changes, enforce constraints, or update related tables.

Example:
We previously created an AFTER UPDATE trigger to log salary changes. Another example could be tracking the number of times an employee's record has been updated.

```sql
CREATE TRIGGER TrackUpdateCount
AFTER UPDATE ON Employees
FOR EACH ROW
BEGIN
   UPDATE Employees
   SET Update_Count = Update_Count + 1
   WHERE Employee_ID = NEW.Employee_ID;
END;
```

Delete Triggers:

A DELETE trigger executes when a row is deleted from the table. It can be used to enforce referential integrity or archive records.

Example:
Let's create a trigger that logs the deletion of any employee record.

```sql
CREATE TRIGGER TrackUpdateCount
AFTER UPDATE ON Employees
FOR EACH ROW
BEGIN
   UPDATE Employees
   SET Update_Count = Update_Count + 1
   WHERE Employee_ID = NEW.Employee_ID;
END;
```

1. **Trigger Name:** LogEmployeeDeletion
2. **AFTER DELETE:** The trigger executes after a DELETE operation on the Employees table.
3. **Trigger Logic:** The trigger inserts a record into the DeletionLog table, capturing the Employee_ID and the deletion date.

Data Modelling and Normalization

Data modelling is the process of defining and structuring data within a database to ensure that it is accurately represented and efficiently stored. The key purpose of data modeling is to provide a clear and organized framework for how data is related, how it can be retrieved, and how it should be stored in a database.

Normalization is a fundamental part of the data modeling process. It involves organizing the columns (attributes) and tables (entities) of a relational database to minimize redundancy and dependency. Normalization typically involves dividing large tables into smaller, more manageable tables and defining relationships between them to improve the data's integrity and consistency.

- **Entity-Relationship (ER) diagrams**

Entity-Relationship (ER) diagrams are a visual representation of the database structure. They illustrate the relationships between different entities (tables) within a database. ER diagrams use symbols to represent entities, attributes, and relationships, providing a clear picture of how data is interconnected.

1. **Entities:** These are objects or concepts about which data is stored. In a database, entities are usually represented as tables.
2. **Attributes:** These are the details or properties of an entity, represented as columns in a table.
3. **Relationships:** These define how entities are related to each other. Relationships are depicted using lines that connect entities in an ER diagram.

Example ER Diagram: Let's consider a simple ER diagram for a university database:

Entities:

Student (attributes: Student_ID, Name, DOB)
Course (attributes: Course_ID, Course_Name, Credits)
Enrollment (attributes: Enrollment_ID, Grade)

Relationships:

A Student can enroll in many Courses (One-to-Many)
A Course can have many Students enrolled (Many-to-One)

This ER diagram would show the relationships between students, courses, and their enrollments.

- **Normalization concepts (1NF, 2NF, 3NF, BCNF)**

Normalization is the process of organizing data in a database into tables to minimize redundancy and ensure data integrity. There are several stages, or "normal forms," each with its own rules and requirements.

First Normal Form (1NF):

Definition: A table is in 1NF if it contains only atomic (indivisible) values and each column contains only one type of data.

Example: Consider a Student table that stores multiple phone numbers in a single column. To convert it to 1NF, you would create a separate row for each phone number.

Original Table (Not in 1NF):

Student_ID	Name	Phone_Numbers
1	John Doe	123-456, 789-012

Normalized Table (1NF):

Student_ID	Name	Phone_Number
1	John Doe	123-456
1	John Doe	789-012

Second Normal Form (2NF):

Definition: A table is in 2NF if it is in 1NF and all non-key attributes are fully dependent on the primary key.

Example: Consider an Enrollment table where each row includes the student's address. The student's address depends only on Student_ID, not on Course_ID. To achieve 2NF, you should separate the Student information into its own table.

Original Table (1NF):

Enrollment_ID	Student_ID	Course_ID	Address
1	1	101	123 Main St

Normalized Table (2NF):

Student Table:

Student_ID	Name	Address
1	John Doe	123 Main St

Enrollment Table:

Enrollment_ID	Student_ID	Course_ID
1	1	101

Third Normal Form (3NF):

Definition: A table is in 3NF if it is in 2NF and all the attributes are dependent only on the primary key and not on any other non-key attribute.

Example: If the Student table contains a City column that depends on the ZIP_Code, this violates 3NF because City depends on ZIP_Code, not Student_ID. To normalize, move City and ZIP_Code into a separate Locations table.

Original Table (2NF):

Student_ID	Name	ZIP_Code	City
1	John Doe	12345	New York

Normalized Table (3NF):

Student Table:

Student_ID	Name	ZIP_Code
1	John Doe	12345

Locations Table:

ZIP_Code	City
12345	New York

Boyce-Codd Normal Form (BCNF):

Definition: BCNF is a stricter version of 3NF. A table is in BCNF if it is in 3NF and every determinant is a candidate key. BCNF ensures that there are no anomalies due to functional dependencies.

Key Concepts:

- **Super Key:** A set of one or more attributes that can uniquely identify a record in a table.
- **Candidate Key:** A minimal super key; a key that can uniquely identify records in the table and contains no unnecessary attributes.
- **Functional Dependency:** A relationship where one attribute uniquely determines another attribute.

Why BCNF is Important:
Even if a table is in 3NF, it might still have anomalies (such as redundancy or insertion/deletion/update anomalies) due to dependencies that are not accounted for by 3NF. BCNF addresses these anomalies by ensuring that every functional dependency in the table involves a super key.

Example of BCNF Violation and Resolution:

Original Table Structure (Not in BCNF):

Student_ID	Course_ID	Instructor_Name
1	101	Prof. Smith
2	101	Prof. Smith
3	102	Prof. Johnson
4	103	Prof. Brown
5	103	Prof. Brown

Functional Dependencies:

1. Student_ID → Course_ID
2. Course_ID → Instructor_Name

Candidate Keys:
Student_ID and Course_ID are both candidate keys.

Analyzing the Table:

Here, Course_ID uniquely determines Instructor_Name.

Student_ID is also a candidate key, but it doesn't determine Instructor_Name on its own.

This table violates BCNF because the Course_ID is not a super key, yet it determines Instructor_Name. This means the table has redundancy and can lead to anomalies.

BCNF Resolution:

To bring this table into BCNF, we must split it into two tables where every functional dependency has a super key on the left-hand side.

Step 1: Identify and Split the Table Based on Dependencies:

1. **Course Table:**
   - Attributes: Course_ID, Instructor_Name
   - Primary Key: Course_ID

Course_ID	Instructor_Name
101	Prof. Smith
102	Prof. Johnson
103	Prof. Brown

2. **Enrollment Table:**
   - Attributes: Student_ID, Course_ID
   - Primary Key: (Student_ID, Course_ID)

Student_ID	Course_ID
1	101
2	101
3	102
4	103
5	103

Step 2: Verify the New Structure:

- In the Course table, Course_ID is a super key and it determines Instructor_Name.
- In the Enrollment table, the combination of Student_ID and Course_ID forms a composite key that uniquely identifies each record

Benefits of BCNF:

Eliminates Redundancy: By splitting the original table, we eliminate redundant storage of the instructor's name for each student enrolled in a course.

Prevents Anomalies: The risk of anomalies (insertion, deletion, and update anomalies) is reduced, ensuring data consistency and integrity.

a. **Insertion Anomaly:** We can add new courses without needing to assign a student.
b. **Deletion Anomaly:** Dropping a student from a course doesn't cause loss of information about the course or instructor.
c. **Update Anomaly:** Changing an instructor's name in the course table automatically reflects for all students without requiring multiple updates.

- **Denormalization and performance considerations**

Denormalization is the process of combining tables that have been normalized into more complex tables. While normalization reduces redundancy, it can also lead to performance issues in large databases due to the need for complex joins across multiple tables. Denormalization is sometimes used to improve read performance by reducing the number of joins needed to retrieve data.

Example: In a highly normalized database, a query to retrieve a student's courses may require joins across several tables. By denormalizing, you might create a single table that stores both student and course data together, thus reducing the need for joins.

Performance Considerations:
Read vs. Write Performance: Normalization improves data integrity and reduces redundancy, which is beneficial for write-heavy systems. However, it can slow down read operations that require joining multiple tables.

Query Optimization: In denormalized tables, query performance can be optimized, especially for read-heavy operations. However, this comes at the cost of potential data redundancy and increased storage requirements.

- **Database design principles**

Effective database design is crucial for ensuring data integrity, efficiency, and scalability. Key principles include:

Consistency: Data should remain consistent across the database, especially during transactions. This can be achieved through ACID properties (Atomicity, Consistency, Isolation, Durability).

Integrity: Referential integrity must be maintained, ensuring that relationships between tables are accurate and that foreign keys match primary keys in related tables.

Flexibility: The database design should be flexible enough to accommodate future changes without requiring significant restructuring.

Scalability: The database should be designed to handle growth in data volume and user load without significant performance degradation.

Example Scenarios:

Normalization Example:
- **Scenario:** You have a table that stores student information along with the courses they are enrolled in. The table also includes the instructor's name for each course.
- **Normalization Process:**
  - **1NF:** Ensure that the table only contains atomic values.
  - **2NF:** Remove partial dependencies by separating student and course information into different tables.
  - **3NF:** Remove transitive dependencies by separating instructor information into another table.

Denormalization Example:

Scenario: A large e-commerce database where customer orders are stored in multiple tables. To improve query performance for reporting, you denormalize by combining customer, order, and product information into a single table.

Result: Queries become faster because fewer joins are required, but data redundancy increases.