Data Modification Statements (DML)

SQL Select:

The SQL `SELECT` statement is used to retrieve data from a database.

Syntax:

```
SELECT column1, column2, ...
FROM table_name;
```

- `SELECT`: Indicates that you want to retrieve data from the specified columns.
- `column1, column2, ...`: Names of the columns you want to retrieve data from. You can specify multiple columns separated by commas.
- `FROM table_name`: Specifies the table from which you want to retrieve the data.

Example:

```
SELECT FirstName, LastName
FROM Employees;
```

Select All:

We can use the asterisk (*) symbol to select all columns from a table.

Syntax:

```
SELECT *
FROM table_name;
```

Example:

```
SELECT *
FROM Employees;
```

Adding Conditions:

You can also add conditions to your `SELECT` statement using the `WHERE` clause to filter the rows returned based on specific criteria.

Example with Condition:
Suppose you want to retrieve the
`FirstName` and `LastName` of employees with a salary greater than $50,000:

```
SELECT FirstName, LastName
FROM Employees
WHERE Salary > 50000;
```

Operators:

Here are the common SQL operators that you can use within the `WHERE` clause to construct conditions:

Comparison Operators:

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

Logical Operators:

- `AND`: Combines two or more conditions. Both conditions must be true for the row to be included.
- `OR`: Combines two or more conditions. At least one condition must be true for the row to be included.
- `NOT`: Negates a condition. It returns true if the condition is false, and vice versa.

Example Usage:

```
SELECT *
FROM Employees
WHERE Salary > 50000
  AND Department = 'Sales';
```

Pattern Matching Operators:

Like Operator:

The `LIKE` operator in SQL is used to search for a specified pattern within a column's value. It's often combined with wildcard characters to match patterns more flexibly. The `%` wildcard character is commonly used with the `LIKE` operator to represent zero or more characters in a pattern.

1. **Starts with:**
   - To match patterns that start with certain characters, you can use the `%` wildcard character at the end of the pattern.
   - **Syntax:**

```
SELECT column1, column2, ...
FROM table_name
WHERE column_name LIKE 'pattern%';
```

- **Example:**

```
SELECT *
FROM Products
WHERE ProductName LIKE 'A%';
```

- This query will retrieve all rows from the `Products` table where the `ProductName` starts with the letter **'A'**.

2. **Ends with:**
   - To match patterns that end with certain characters, you can use the `%` wildcard character at the beginning of the pattern.
   - **Syntax:**

```
SELECT column1, column2, ...
FROM table_name
WHERE column_name LIKE '%pattern';
```

   - **Example:**

```
SELECT *
FROM Customers
WHERE Email LIKE '%gmail.com';
```

   - This query will retrieve all rows from the `Customers` table where the `Email` ends with **'@gmail.com'**.
3. **Contains:**
   - To match patterns that contain a specific substring, you can use the `%` wildcard character both before and after the pattern.
   - **Syntax:**

```
SELECT column1, column2, ...
FROM table_name
WHERE column_name LIKE '%pattern%';
```

   - **Example:**

```
SELECT *
FROM Products
WHERE ProductName LIKE '%apple%';
```

- This query will retrieve all rows from the `Products` table where the `ProductName` contains the substring **'apple'**.

Between Operator:

The `BETWEEN` operator in SQL is used to specify a range of values for comparison in a column. It allows you to specify a lower and upper bound, inclusively.

- **Syntax:**

```
SELECT column1, column2, ...
FROM table_name
WHERE column_name BETWEEN value1 AND value2;
```

- **Example:**

```
SELECT *
FROM Products
WHERE Price BETWEEN 10 AND 50;
```

- This query will retrieve all rows from the `Products` table where the `Price` column falls within the range of **10 to 50**.

IN Operator:

The `IN` operator in SQL is used to specify multiple values for comparison in a column. It allows you to specify a list of values to match against.

- **Syntax:**

```
SELECT column1, column2, ...
FROM table_name
WHERE column_name IN (value1, value2, ...);
```

- **Example:**

```
SELECT *
FROM Orders
WHERE CustomerID IN (1, 2, 3);
```

- This query will retrieve all rows from the `Orders` table where the `CustomerID` column matches any of the specified values (1, 2, or 3).

Select Distinct:

In SQL, the `SELECT DISTINCT` statement is used to retrieve unique values from a specific column or combination of columns in a table.

- **Syntax:**

```
SELECT DISTINCT column1, column2, ...
FROM table_name;
```

- **Example:**

```
SELECT DISTINCT Department
FROM Employees;
```

Select As Alias:

In SQL, the `AS` keyword is used to assign an alias to a column or an expression in the result set of a `SELECT` query. Aliases are temporary names assigned to columns or expressions in the result set, which can be useful for readability or when using the result set in subsequent calculations or queries.

- **Syntax:**

```
SELECT column_name AS alias_name
FROM table_name;
```

- **Example:**

```
SELECT FirstName AS First_Name, LastName AS Last_Name
FROM Employees;
```

- This query will select the `FirstName` column from the `Employees` table and rename it as `First_Name`, and it will select the `LastName` column and rename it as `Last_Name` in the result set.

Concatenate Columns:

In SQL, you can concatenate (combine) the values of multiple columns into a single column using the `CONCAT()` function. This allows you to create a new column that contains the concatenated values of the specified columns.

- **Syntax:**

```
SELECT CONCAT(column1, ' ', column2) AS concatenated_column
FROM table_name;
```

- **Example:**

```
SELECT CONCAT(FirstName, ' ', LastName) AS Full_Name
FROM Employees;
```

This query will concatenate the `FirstName` and `LastName` columns from the `Employees` table with a space in between and rename the result as `Full_Name` in the result set.

Select Top:

In SQL, the `SELECT TOP` clause is used to retrieve a specified number of rows from the beginning of a result set. This clause is often used to limit the number of rows returned by a query, especially when dealing with large datasets.

- **Syntax:**

```
SELECT TOP number column1, column2, ...
FROM table_name
WHERE condition;
```

- **Example:**

```
SELECT TOP 5 FirstName, LastName
FROM Customers
```

Ordered By:

In SQL, the `ORDER BY` clause is used to sort the result set of a query based on one or more columns. It allows you to specify the order in which the rows should appear in the result set.

- **Syntax:**

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1 [ASC|DESC], column2 [ASC|DESC], ...;
```

- **Example:**

```
SELECT FirstName, LastName, Age
FROM Employees
ORDER BY LastName ASC, FirstName ASC;
```

This query retrieves the `FirstName`, `LastName`, and `Age` columns from the `Employees` table and sorts the result set first by `LastName` in ascending order and then by `FirstName` in ascending order.

Create table from an existing table:

To create a new table from an existing table in SQL, you can use the `SELECT INTO` statement.

- **Syntax:**

```
SELECT column1, column2, ...
INTO new_table
FROM existing_table
WHERE condition;
```

- **Example:**

```
SELECT CustomerID, FirstName, LastName, Email
INTO new_customers
FROM customers
WHERE Country = 'USA';
```

  This query creates a new table `new_customers` with the same columns as the `customers` table, but only for customers from the USA.

SQL Insert:

The `INSERT INTO` statement in SQL is used to add new records (rows) to a table. It allows you to specify the table name and the values you want to insert into the columns.

- **Syntax:**

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

- **Example:**

```
INSERT INTO Customers (CustomerID, FirstName, LastName, Email)
VALUES (1, 'John', 'Doe', 'john.doe@example.com');
```

  This query inserts a new record into the `Customers` table with specific values for the `CustomerID`, `FirstName`, `LastName`, and `Email` columns.

- If you want to insert values for all columns, you can omit the column list:

```
INSERT INTO Products
VALUES (101, 'Product A', 25.99, 50);
```

  This query inserts a new record into the `Products` table without specifying column names. The values must be in the same order as the columns in the table.

- You can also insert multiple records in a single `INSERT INTO` statement:

```
INSERT INTO Orders (CustomerID, ProductID, Quantity)
VALUES (1, 101, 3),
       (2, 102, 5),
       (1, 103, 2);
```

This query inserts three new records into the `Orders` table with different combinations of `CustomerID`, `ProductID`, and `Quantity`.

SQL Insert Into Select:

The `INSERT INTO SELECT` statement in SQL is used to insert data from one table into another table. It allows you to copy data from one table (or a result set of a query) and insert it into another table, potentially with modifications or filtering.

- **Syntax:**

```
INSERT INTO target_table (column1, column2, ...)
SELECT column1, column2, ...
FROM source_table
WHERE condition;
```

- **Example:**

```
INSERT INTO NewCustomers (FirstName, LastName, Email)
SELECT FirstName, LastName, Email
FROM OldCustomers
WHERE RegistrationDate > '2022-01-01';
```

This query inserts data into the `NewCustomers` table from the `OldCustomers` table, but only for customers who registered after January 1, 2022.

SQL Update:

The `UPDATE` statement in SQL is used to modify existing records (rows) in a table. It allows you to change the values of one or more columns for one or more rows based on specified conditions.

- **Syntax:**

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

- **Example:**

```
UPDATE Customers
SET FirstName = 'Jane', LastName = 'Smith'
WHERE CustomerID = 1;
```

This query updates the `FirstName` and `LastName` columns of the customer with `CustomerID` equal to 1 in the `Customers` table.

- To update all records in a table, you can omit the `WHERE` clause:

```
UPDATE Orders
SET Status = 'Shipped';
```

This query updates the `Status` column of all records in the `Orders` table to 'Shipped'.

SQL Delete:

The `DELETE` statement in SQL is used to remove one or more records (rows) from a table based on specified conditions.

- **Syntax:**

```
DELETE FROM table_name
WHERE condition;
```

- **Example:**

```
DELETE FROM Customers
WHERE CustomerID = 1;
```

This query deletes the record with `CustomerID` equal to 1 from the `Customers` table.

- You can also delete all records in a table by omitting the `WHERE` clause:

```
DELETE FROM Products;
```

This query deletes all records from the `Products` table.

- Additionally, you can use subqueries in the `WHERE` clause to specify more complex conditions for deletion:

```
DELETE FROM Orders
WHERE CustomerID IN (SELECT CustomerID FROM Customers WHERE Country = 'US
A');
```

This query deletes all orders associated with customers from the USA.

Aggregate Functions:

In SQL, aggregate functions such as `MIN`, `MAX`, `COUNT`, `SUM`, and `AVG` are used to perform calculations on a set of values in a column.

- **MIN:** Returns the smallest value in a column.

```
SELECT MIN(column_name)
FROM table_name;
```

- **MAX:** Returns the largest value in a column.

```
SELECT MAX(column_name)
FROM table_name;
```

- **COUNT:** Returns the number of rows in a table or the number of non-null values in a column.

```
SELECT COUNT(*)
FROM table_name; -- Count all rows

SELECT COUNT(column_name)
FROM table_name; -- Count non-null values in a column

SELECT COUNT(DISTINCT column_name)
FROM table_name; -- Count the number of unique rows
```

- **SUM:** Returns the total sum of values in a column.

```
SELECT SUM(column_name)
FROM table_name;
```

- **AVG:** Returns the average value of a column.

```
SELECT AVG(column_name)
FROM table_name;
```

Set Operations:

In SQL, `UNION`, `UNION ALL`, `INTERSECT`, and `EXCEPT` are set operations used to combine or compare the result sets of two or more queries.

- **UNION:**
  - Combines the result sets of two or more SELECT statements into a single result set.
  - Removes duplicate rows from the result set.

- **Syntax:**

```
SELECT column1, column2, ...
FROM table1
UNION
SELECT column1, column2, ...
FROM table2;
```

```
SELECT name
FROM Teachers
UNION
SELECT name
FROM Students;
```

Here, the SQL command selects the union of the name columns from two different tables: Teachers and Students.

- **UNION ALL:**
  - Combines the result sets of two or more SELECT statements into a single result set.
  - Retains all rows, including duplicates, from the combined result sets.
  - Syntax:

```
SELECT column1, column2, ...
FROM table1
UNION ALL
SELECT column1, column2, ...
FROM table2;
```

- **INTERSECT:**
  - Returns the intersection of two result sets, i.e., only the rows that appear in both result sets.
  - Does not remove duplicate rows.
  - Syntax:

```
SELECT column1, column2, ...
FROM table1
INTERSECT
SELECT column1, column2, ...
FROM table2;
```

- **EXCEPT:**
  - Returns the set difference of two result sets, i.e., only the rows that appear in the first result set but not in the second result set.

- Does not remove duplicate rows.
- Syntax:

```
SELECT column1, column2, ...
FROM table1
EXCEPT
SELECT column1, column2, ...
FROM table2;
```

Is Null Operator:

The `IS NULL` operator in SQL is used to check if a value in a column is NULL. NULL represents the absence of a value, and it is different from an empty string or zero. The `IS NULL` operator returns true if the value is NULL and false otherwise.

Here's the syntax for using the `IS NULL` operator:

```
SELECT column1, column2, ...
FROM table_name
WHERE column_name IS NULL;
```

And here's an example of how the `IS NULL` operator can be used:

```
SELECT first_name, last_name
FROM employees
WHERE middle_name IS NULL;
```

Similarly, you can use the `IS NOT NULL` operator to check if a value is not NULL.

```
SELECT column1, column2, ...
FROM table_name
WHERE column_name IS NOT NULL;
```

This query selects rows where the value in `column_name` is not NULL.

Nested Query:

It is a query embedded within another SQL query. It allows you to use the result of one query as a condition or value in another query.

Syntax:

```
SELECT column1, column2, ...
FROM table1
```

```
WHERE column_name operator (SELECT column_name FROM table2 WHERE condition);
```

Example:

```
SELECT first_name
FROM Customers
WHERE age= (
    -- subquery
    SELECT MAX(age)
    FROM CUSTOMERS
);
```

Here, the query is divided into two parts:

- the subquery selects the maximum **age** from the **Customers** table
- the outer query selects the **first_name** of the customer with the maximum **id** (returned by the subquery)

Exists Operator:

The `EXISTS` operator in SQL is used to check whether a subquery returns any rows. It returns true if the subquery returns one or more rows, and false if the subquery returns no rows. The `EXISTS` operator is commonly used in conjunction with a correlated subquery to test for the existence of certain conditions.

Syntax:

```
SELECT column1, column2, ...
FROM table_name
WHERE EXISTS (subquery);
```

Example:

```
SELECT customer_id, first_name
FROM Customers
WHERE EXISTS (
  SELECT order_id
  FROM Orders
  WHERE Orders.customer_id = Customers.customer_id
);
```

Any and All Operators:

The `ANY` and `ALL` operators in SQL are used in combination with subqueries to compare a value with a set of values returned by the subquery. These operators are typically used in combination with

comparison operators such as `=`, `>`, `<`, etc.

ANY Operator:

- The `ANY` operator returns `TRUE` if the comparison is true for at least one of the values returned by the subquery.
- **Syntax:**

```
SELECT column_name
FROM table_name
WHERE column_name operator ANY (subquery);
```

- **Example:**

```
SELECT *
FROM Teachers
WHERE age = ANY (
  SELECT age
  FROM Students
);
```

ALL Operator:

- The `ALL` operator returns `TRUE` if the comparison is true for all values returned by the subquery.
- **Syntax:**

```
SELECT column_name
FROM table_name
WHERE column_name operator ALL (subquery);
```

- **Example:**

```
SELECT *
FROM Teachers
WHERE age >  ALL (
  SELECT age
  FROM Students
);
```

Join:

In SQL, a JOIN operation is used to combine rows from two or more tables based on a related column between them. Joins are fundamental for retrieving data from multiple tables in a relational database. There are different types of joins:

1. **INNER JOIN:** Returns only the rows that have matching values in both tables.
   - **Syntax:**

     ```
     SELECT columns
     FROM table1
     INNER JOIN table2 ON table1.column_name = table2.column_name;
     ```

   - **Example:**
     This query selects the first name of employees along with the name of their respective departments by joining the `employees` table with the `departments` table on the `department_id` column.

     ```
     SELECT employees.first_name, departments.department_name
     FROM employees
     INNER JOIN departments ON employees.department_id = departments.departm
     ent_id;
     ```

   NOTE: We can also use SQL JOIN instead of INNER JOIN. Basically, these two clauses are the same.
2. **LEFT JOIN (or LEFT OUTER JOIN):** Returns all rows from the left table and the matched rows from the right table. If there is no match, NULL values are returned for columns from the right table.
   - **Syntax:**

     ```
     SELECT columns
     FROM table1
     LEFT JOIN table2 ON table1.column_name = table2.column_name;
     ```

   - **Example:**
     This query selects the first name of employees along with the name of their respective departments. All employees are returned, even if they do not belong to any department.

     ```
     SELECT employees.first_name, departments.department_name
     FROM employees
     LEFT JOIN departments ON employees.department_id = departments.departme
     nt_id;
     ```

3. **RIGHT JOIN (or RIGHT OUTER JOIN):** Returns all rows from the right table and the matched rows from the left table. If there is no match, NULL values are returned for columns from the left table.
   - **Syntax:**

     ```
     SELECT columns
     FROM table1
     ```

```
RIGHT JOIN table2 ON table1.column_name = table2.column_name;
```

- **Example:**
  This query selects the first name of employees along with the name of their respective departments. All departments are returned, even if they do not have any employees.

```
SELECT employees.first_name, departments.department_name
FROM employees
RIGHT JOIN departments ON employees.department_id = departments.departm
ent_id;
```

4. **FULL JOIN (or FULL OUTER JOIN):** Returns all rows when there is a match in either the left or right table. If there is no match, NULL values are returned for columns from the other table.

   - **Syntax:**

```
SELECT columns
FROM table1
FULL JOIN table2 ON table1.column_name = table2.column_name;
```

   - **Example:**
     This query selects the first name of employees along with the name of their respective departments. All employees and departments are returned, with NULL values for unmatched rows.

```
SELECT employees.first_name, departments.department_name
FROM employees
FULL JOIN departments ON employees.department_id = departments.departme
nt_id;
```

In SQL, there are several other types of joins that serve different purposes:

1. **Self Join:** A self join is a join that occurs when a table is joined with itself. It is useful when you want to compare rows within the same table.

   - **Syntax:**

```
SELECT t1.column1, t2.column2
FROM table t1
JOIN table t2 ON t1.related_column = t2.related_column;
```

   - **Example:**
     This query selects the first name of employees along with the first name of their respective managers by joining the `employees` table with itself on the `manager_id` column.

```
SELECT e1.first_name, e2.first_name AS manager_name
FROM employees e1
JOIN employees e2 ON e1.manager_id = e2.employee_id;
```

2. **Natural Join:** A natural join is a join that automatically joins two tables based on columns with the same name. It does not require specifying the columns to join explicitly.
   - **Syntax:**

     ```
     SELECT *
     FROM table1
     NATURAL JOIN table2;
     ```

   - **Example:**
     This query automatically joins the `employees` table with the `departments` table based on columns with the same name, such as `department_id`.

     ```
     SELECT *
     FROM employees
     NATURAL JOIN departments;
     ```

Group By:

In SQL, the `GROUP BY` clause is used to group rows that have the same values into summary rows, typically for use with aggregate functions (such as `COUNT`, `SUM`, `AVG`, `MAX`, `MIN`) to perform operations on each group.

Here's the basic syntax of the `GROUP BY` clause:

```
SELECT column1, aggregate_function(column2)
FROM table_name
GROUP BY column1;
```

- `column1`: The column(s) by which you want to group the result set.
- `aggregate_function`: The aggregate function to perform on each group.

And here's an example to illustrate how `GROUP BY` works:

Let's say we have a table named `orders` with columns `product_id`, `customer_id`, and `quantity`.

To count the number of orders for each product, you can use `GROUP BY` with the `COUNT` aggregate function:

```
SELECT product_id, COUNT(*) AS num_orders
FROM orders
```

```
GROUP BY product_id;
```

In this example:

- We specify `product_id` as the column to group the data by.
- We use the `COUNT(*)` aggregate function to count the number of rows in each group.
- The result will show the `product_id` along with the count of orders (`num_orders`) for each product.

Additionally, you can use `GROUP BY` with multiple columns to create groups based on multiple criteria. For example:

```
SELECT product_id, customer_id, SUM(quantity) AS total_quantity
FROM orders
GROUP BY product_id, customer_id;
```

This query groups the data by both `product_id` and `customer_id`, and calculates the total quantity of each product ordered by each customer.

Having Clause:

In SQL, the `HAVING` clause is used in combination with the `GROUP BY` clause to filter the result set based on aggregated values. While the `WHERE` clause filters individual rows before the grouping operation, the `HAVING` clause filters groups of rows after the grouping operation.

Here's the basic syntax of the `HAVING` clause:

```
SELECT column1, aggregate_function(column2)
FROM table_name
GROUP BY column1
HAVING condition;
```

- `column1`: The column(s) by which you want to group the result set.
- `aggregate_function`: The aggregate function to perform on each group.
- `condition`: The condition to filter groups.

And here's an example to illustrate how `HAVING` works:

Suppose we have a table named `orders` with columns `product_id`, `customer_id`, and `quantity`.

To find products with total orders greater than a certain threshold (let's say 100 units), you can use `GROUP BY` with `HAVING`:

```
SELECT product_id, SUM(quantity) AS total_quantity
FROM orders
GROUP BY product_id
HAVING SUM(quantity) > 100;
```

In this example:

- We group the data by `product_id` using `GROUP BY`.
- We calculate the total quantity of each product using the `SUM` aggregate function.
- We filter the result set using `HAVING` to only include groups where the total quantity is greater than 100.

Case:

In SQL, the `CASE` expression is used to perform conditional logic within a query. It allows you to specify conditions and return different values based on those conditions. The `CASE` expression can be used in various parts of a SQL query, such as the `SELECT` statement, `WHERE` clause, `ORDER BY` clause, and more.

Here's the basic syntax of the `CASE` expression:

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

- `condition1`, `condition2`, etc.: Conditions that you want to evaluate.
- `result1`, `result2`, etc.: Values to return if the corresponding condition is true.
- `default_result`: Value to return if none of the conditions are true (optional).

And here's an example to illustrate how the `CASE` expression works:

Suppose we have a table named `employees` with columns `first_name`, `last_name`, and `salary`.

To categorize employees based on their salary ranges, you can use a `CASE` expression within a `SELECT` statement:

```
SELECT first_name, last_name,
    CASE
        WHEN salary >= 50000 AND salary < 70000 THEN 'Low'
        WHEN salary >= 70000 AND salary < 90000 THEN 'Medium'
        WHEN salary >= 90000 THEN 'High'
        ELSE 'Unknown'
    END AS salary_category
FROM employees;
```

In this example:

- We use the `CASE` expression to categorize employees into different salary categories.
- If an employee's salary falls within a specific range, the corresponding category is returned.

- If the salary does not fall within any defined range, the 'Unknown' category is returned.

In SQL, the `CASE` expression can also be used in an `UPDATE` statement to conditionally update values in a table based on specified criteria.

Here's the basic syntax of using `CASE` in an `UPDATE` statement:

```
UPDATE table_name
SET column_name =
    CASE
        WHEN condition1 THEN value1
        WHEN condition2 THEN value2
        ...
        ELSE default_value
    END
WHERE condition;
```

```
UPDATE employees
SET salary_grade =
    CASE
        WHEN salary >= 50000 AND salary < 70000 THEN 'Low'
        WHEN salary >= 70000 AND salary < 90000 THEN 'Medium'
        WHEN salary >= 90000 THEN 'High'
        ELSE 'Unknown'
    END;
```

In this example:

- We use the `CASE` expression to determine the appropriate salary grade based on the salary of each employee.
- If an employee's salary falls within a specific range, the corresponding salary grade is assigned.
- If the salary does not fall within any defined range, the 'Unknown' salary grade is assigned.

View Table:

In SQL, a view is a virtual table that is based on the result set of a SELECT query. It is a saved SQL query that can be treated like a table, allowing you to retrieve and manipulate data from it just like you would with a regular table.

Here's how you create a view in SQL:

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

- `view_name`: The name of the view you want to create.
- `column1`, `column2`, etc.: The columns you want the view to include.
- `table_name`: The name of the table(s) from which you want to retrieve data.
- `condition`: Optional condition to filter the data.

```
CREATE VIEW high_salary_employees AS
SELECT first_name, last_name
FROM employees
WHERE salary > 50000;
```

Once the view is created, you can query it like a regular table:

```
SELECT * FROM high_salary_employees;
```

You can also drop view:

```
DROP VIEW high_salary_employees;
```

Stored Procedures:

Stored procedures in SQL Server are precompiled collections of one or more SQL statements that are stored under a name and processed as a unit.

Basic Stored Procedure

A basic stored procedure is created using the `CREATE PROCEDURE` statement. Here's an example:

```
CREATE PROCEDURE GetEmployeeDetails
AS
BEGIN
    SELECT EmployeeID, FirstName, LastName, Department
    FROM Employees
END
```

This stored procedure, `GetEmployeeDetails`, retrieves all employee details from the `Employees` table. To execute this procedure, you would use the `EXEC` statement:

```
EXEC GetEmployeeDetails
```

Parameterized Stored Procedure

Parameterized stored procedures allow you to pass parameters to the stored procedure, making it more flexible and dynamic. Parameters can be input parameters, output parameters, or both.

Input Parameters

Here's an example of a stored procedure with input parameters:

```
CREATE PROCEDURE GetEmployeeByID
    @EmployeeID INT
AS
BEGIN
    SELECT EmployeeID, FirstName, LastName, Department
    FROM Employees
    WHERE EmployeeID = @EmployeeID
END
```

In this example, `@EmployeeID` is an input parameter. To execute this procedure with a specific employee ID, you would do:

```
EXEC GetEmployeeByID @EmployeeID = 1
```

Output Parameters

Stored procedures can also have output parameters to return values. Here's an example:

```
CREATE PROCEDURE GetEmployeeDepartment
    @EmployeeID INT,
    @Department VARCHAR(50) OUTPUT
AS
BEGIN
    SELECT @Department = Department
    FROM Employees
    WHERE EmployeeID = @EmployeeID
END
```

To call this stored procedure and get the department for a specific employee, you would declare a variable to hold the output value:

```
DECLARE @Dept VARCHAR(50)

EXEC GetEmployeeDepartment @EmployeeID = 1, @Department = @Dept OUTPUT

SELECT @Dept AS Department
```

In this example, `@Dept` will hold the value of the department after the stored procedure execution.

Triggers:

Triggers in SQL Server are special types of stored procedures that automatically execute or "trigger" when certain events occur in the database. Triggers can be set to execute in response to data modifications such as `INSERT`, `UPDATE`, or `DELETE` operations.

Let's create comprehensive examples of `AFTER` and `INSTEAD OF` triggers for `INSERT`, `UPDATE`, and `DELETE` operations, as well as an example of a `DATABASE` trigger.

1. Database Setup

First, we'll create the necessary tables:

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName NVARCHAR(50),
    LastName NVARCHAR(50),
    Department NVARCHAR(50)
);

CREATE TABLE EmployeeAudit (
    AuditID INT IDENTITY(1,1) PRIMARY KEY,
    EmployeeID INT,
    FirstName NVARCHAR(50),
    LastName NVARCHAR(50),
    Department NVARCHAR(50),
    Action NVARCHAR(50),
    ActionDate DATETIME DEFAULT GETDATE()
);

CREATE TABLE ArchivedEmployees (
    EmployeeID INT PRIMARY KEY,
    FirstName NVARCHAR(50),
    LastName NVARCHAR(50),
    Department NVARCHAR(50),
    ArchivedDate DATETIME DEFAULT GETDATE()
);
```

2. AFTER Triggers

AFTER INSERT Trigger

```
CREATE TRIGGER trgAfterInsertOnEmployees
ON Employees
AFTER INSERT
AS
```

```
BEGIN
    INSERT INTO EmployeeAudit (EmployeeID, FirstName, LastName, Department, A
ction)
    SELECT EmployeeID, FirstName, LastName, Department, 'INSERT'
    FROM inserted;
END;
```

AFTER UPDATE Trigger

```
CREATE TRIGGER trgAfterUpdateOnEmployees
ON Employees
AFTER UPDATE
AS
BEGIN
    INSERT INTO EmployeeAudit (EmployeeID, FirstName, LastName, Department, A
ction)
    SELECT EmployeeID, FirstName, LastName, Department, 'UPDATE'
    FROM inserted;
END;
```

AFTER DELETE Trigger

```
CREATE TRIGGER trgAfterDeleteOnEmployees
ON Employees
AFTER DELETE
AS
BEGIN
    INSERT INTO EmployeeAudit (EmployeeID, FirstName, LastName, Department, A
ction)
    SELECT EmployeeID, FirstName, LastName, Department, 'DELETE'
    FROM deleted;
END;
```

3. INSTEAD OF Triggers

INSTEAD OF INSERT Trigger

```
CREATE TRIGGER trgInsteadOfInsertOnEmployees
ON Employees
INSTEAD OF INSERT
AS
BEGIN
    -- Perform any custom logic before the insert
```

```
    INSERT INTO Employees (EmployeeID, FirstName, LastName, Department)
    SELECT EmployeeID, FirstName, LastName, Department
    FROM inserted;

    INSERT INTO EmployeeAudit (EmployeeID, FirstName, LastName, Department, A
ction)
    SELECT EmployeeID, FirstName, LastName, Department, 'INSERT'
    FROM inserted;
END;
```

INSTEAD OF UPDATE Trigger

```
CREATE TRIGGER trgInsteadOfUpdateOnEmployees
ON Employees
INSTEAD OF UPDATE
AS
BEGIN
    -- Perform any custom logic before the update
    UPDATE Employees
    SET FirstName = inserted.FirstName, LastName = inserted.LastName, Departm
ent = inserted.Department
    FROM Employees
    INNER JOIN inserted ON Employees.EmployeeID = inserted.EmployeeID;

    INSERT INTO EmployeeAudit (EmployeeID, FirstName, LastName, Department, A
ction)
    SELECT EmployeeID, FirstName, LastName, Department, 'UPDATE'
    FROM inserted;
END;
```

INSTEAD OF DELETE Trigger

```
CREATE TRIGGER trgInsteadOfDeleteOnEmployees
ON Employees
INSTEAD OF DELETE
AS
BEGIN
    -- Archive the data before deleting
    INSERT INTO ArchivedEmployees (EmployeeID, FirstName, LastName, Departmen
t)
    SELECT EmployeeID, FirstName, LastName, Department
    FROM deleted;

    DELETE FROM Employees
```

```
    WHERE EmployeeID IN (SELECT EmployeeID FROM deleted);

    INSERT INTO EmployeeAudit (EmployeeID, FirstName, LastName, Department, A
ction)
    SELECT EmployeeID, FirstName, LastName, Department, 'DELETE'
    FROM deleted;
END;
```

4. DDL Triggers on the Database

DDL Trigger for Schema Changes

```
CREATE TRIGGER trg
ON DATABASE
FOR CREATE_TABLE, ALTER_TABLE, DROP_TABLE
AS
BEGIN
    PRINT 'Hello'
END;
```