→ **Views**

. In SQL, a view is a virtual table that is derived from one or more existing tables
It's a saved SQL query that can be treated and used like a table.
Views provide a way to simplify complexe queries encapsulate logic, and present a customized or summarized view of the underlying data.

Examples:

1) Let's say we have two table : 'Customers' and 'Orders'

Customers [customer_id, customerName, city]
Orders [order_id, Customer_id, product]

Now let's create a view called 'OrderDetails' that combine data from the Customers and Orders tables to show the order details with customer names

```
CREATE VIEW OrderDetails AS
SELECT o.order_id, c.customerName, o.product
FROM Orders o
JOIN Customers c USING (customer_id);
```

The "OrderDetails" view is created based on the SQL query provided.
Once the view is created, we can query it just like any other table:

```
SELECT *
FROM OrderDetails;
```

2) Suppose we have a table 'Person'

Person [name, age, id, income]

We want to know wich age has the highest total income.

So we have to solutions for this problem

↳ One solution is to use nesting in the having clause:

```
SELECT age
FROM person
GROUP BY age
HAVING SUM(income) >= ALL (SELECT SUM (income)
                           FROM person
                           GROUP BY age )
```

↳ Another solution is to create a view

```
CREATE VIEW age_income AS
SELECT age, SUM (Income) AS sumIncome
FROM person
GROUPBY age ;

SELECT age
FROM age_income
WHERE sumIncome = (SELECT MAX (sumIncome)
                   FROM ageIncome)
```

. A view by default is not updatable

So if, we want to insert any data to view, we have 2 solution

1. Change the underlying tables, so that the view will be automatically updated
2. Use trigger to handle it

- Views offer several advantages:
  1. Data abstraction: Views allow us to hide complex underlying queries and present a simpler and more intuitive representation of the data
  2. Security: Views can restrict access to certain columns or rows, providing an additional layer of security by controlling the data that users can see or modify
  3. Query reusability: Views encapsulate commonly used queries, enabling us to reuse them across different part of the application
  4. Performance optimization: Views can precompute aggregation, filtering, or join operations, improving query performance by reducing the need for redundant computations and useful in speeding up query answering

→ Triggers

- Triggers are special database objects that are designed to automatically execute a set of actions or statements when a specific event occurs in the database

- It's a code of block that automatically gets executed before or after an insert, update, or delete statement

- Triggers are useful for enforcing business rule, maintaining data integrity and automating tasks within a database system

→ Trigger Types

. Insert Trigger : Executes when a new record is inserted into a table
. Update Trigger : Executes when an existing record is updated
. Delete Trigger : Executes when a record is deleted

→ Trigger structure

. Event: Specifies the event that will execute the trigger (INSERT, UPDATE, DELETE)

. Trigger Action: Contains the action or statements to be executed

. Triggering Time: Determines when the trigger is executed (BEFORE, or AFTER the event)

. Trigger Scope: Specifies whether the trigger operates on each row affected or on the entire set of affected rows.

→ Syntax:

```
CREATE TRIGGER <trigger_name>
[BEFORE | AFTER] <event> ON <table>
<trigger_type>
BEGIN
    <trigger_body>
END;
```

Determine when trigger occurs in response to a specific event

The table which the trigger binds

Could be INSERT, UPDATE, or DELETE

Specify the type of trigger using either FOR EACH ROW or FOR EACH STATEMENT

The code to be executed

→ Examples :

1) Insert trigger

Suppose we have a table called "Employees"

Employees [ID, Name, Salary]

We want to create a trigger to automatically update a separate table "Audit" whenever a new employee is inserted

```
DELIMITER $$
CREATE TRIGGER after_employee_insert
AFTER INSERT ON Employees
FOR EACH ROW
BEGIN
    INSERT INTO Audit (Event, EmployeeID)
    VALUES ('Employee Insert', NEW.ID);
END
$$
DELIMITER ;
```

2) TRIGGER on Update

Let's say we have a table called "Orders"

Orders [orderID, Product, Quantity]

We want to ensure that whenever the quantity of an order is updated, the corresponding product's inventory is adjusted.

```
CREATE TRIGGER after_order_update
AFTER UPDATE ON Orders
FOR EACH ROW
BEGIN
    UPDATE Inventory
    SET Stock = Stock - (NEW.Quantity - old.Quantity)
    WHERE Product = NEW.Product;
END;
```

3) Modify a trigger

```
CREATE OR REPLACE TRIGGER ...
```

4) Drop a trigger

```
DROP TRIGGER IF EXIST table_name.trigger_name;
```

or

```
DROP TRIGGER IF EXIST trigger_name;
```

→ Trigger Usage

. Log table modifications

↳ Some table have sensitive data (e.g. customer email, employee salary) that all changes must be logged - need the UPDATE trigger to insert the changes into a separate log table

. Enforce complexe integrity of data

↳ Define triggers to validate the data and reformat the data before inserting or updating

→ Stored Procedures

. A stored procedure is a prepared SQL code that is stored in the database and can be executed multiple times

. They allow us to encapsulate complex querres making it easier to manage and maintain our code

1. Creation of a stored Procedure

```
CREATE PROCEDURE procedure_name
AS
BEGIN
    SQL_query
END;
```

2. Execute a Stored Procedure

```
EXECUTE Procedure_name;
```

Example:

```
CREATE PROCEDURE GetCustomer
AS
BEGIN
    SELECT * FROM customers;
END;

EXECUTE GetCustomer;
```

3. Parameters

- Stored Procedures can have parameters that allows us to pass values to the procedure
- These values will be processed or returned via the execution of the procedure
- There are three types of parameters:
  IN, OUT and INOUT

Example:

```
CREATE PROCEDURE selectCustomers
@City varchar(30)
AS
BEGIN
SELECT *
FROM customers
WHERE City = @City
END;

EXECUTE selectCustomers @City = 'London';
```

→ Functions

. In SQL Server, user-defined Functions (UDF) are used to save SQL statement permanently in the system.
. UDF are like containers for SQL statements
. They accept input parameters, perform actions and return a result set or a single value

Syntaxe

```
CREATE FUNCTION function-name
(parameter list)
RETURN datatype
IS
BEGIN
    <body>
    RETURN (return-value);
END;
```

→ Difference between Procedures and Triggers

. Both Procedures and Triggers can perform SQL statements
. However, the key difference lies in how they are executed.
. Stored Procedures require manual execution by a user while triggers are automatically executed by the Systeme in response to specific events.
. These events, such as data inserts, update, or delete activate triggers
. A drawback of triggers is that they cannot accept parameters, unlike stored procedures

→ Difference between Procedures and Functions

. Both stored Procedures and Functions are database objects which contain a set of SQL statements to complete a task
. However they are different from each other in many ways.
. Stored procedures are pre-compiled objects which are compiled for the first time and its compiled format is saved, which executes (compiled mode) whenever it is called
. A Function is compiled and execute every time whenever it is called. A Function must return a value and cannot modify the data received as parameters

→ So the difference between Functions and Procedure is:

. Functions must return a value but in stored procedure it is optional. Even a procedure can return return 0 or n values
. Function can have only input parameters while procedures can have input or output parameters
. Functions have limited Functionality compared to stored procedures. They are suitable when we don't need to make permanent changes to the databases objects because we cannot modify the database trough functions
. Unlike procedures, Functions are not stand-alone executables. They need to be used within a context, such as assigning their output to a variable or using them in a Select statement