UNIT II – Relational Model

Relational Data Model - keys, referential integrity and foreign keys, Relational Algebra - SQL fundamentals- Introduction, data definition in SQL, table, key and foreign key definitions, update behaviors-Views, Triggers, Joins, Constraints, Stored Procedure- Intermediate SQL-Advanced SQL features -Embedded SQL- Dynamic SQL

Views

A view in SQL is a saved SQL query that acts as a virtual table. It can fetch data from one or more tables and present it in a customized format, allowing developers to:

- **Simplify Complex Queries:** Encapsulate complex joins and conditions into a single object.
- **Enhance Security:** Restrict access to specific columns or rows.
- **Present Data Flexibly:** Provide tailored data views for different users.

Syntax

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

Multiple Table

```
CREATE VIEW MarksView AS
SELECT StudentDetails.NAME, StudentDetails.ADDRESS, StudentMarks.MARKS
FROM StudentDetails, StudentMarks
WHERE StudentDetails.NAME = StudentMarks.NAME;
```

Rules

- The SELECT statement which is used to create the view should not include GROUP BY clause or ORDER BY clause.
- The SELECT statement should not have the DISTINCT keyword.
- The View should have all NOT NULL values.
- The view should not be created using nested queries or complex queries.
- The view should be created from a single table. If the view is created using multiple tables then we will not be allowed to update the view.

Triggers

SQL triggers are a critical feature in database management systems (DBMS) that provide automatic execution of a set of SQL statements when specific database events, such as INSERT, UPDATE, or DELETE operations, occur. Triggers are commonly used to maintain data integrity, track changes, and enforce business rules automatically, without needing manual input.

A trigger is a stored procedure in a **database** that automatically invokes whenever a special event in the database occurs. By using **SQL triggers**, developers can **automate tasks**, ensure **data consistency**, and keep accurate records of **database activities**.

Example: a trigger can be invoked when a row is inserted into a specified table or when specific table columns are updated.

Syntax:

```
create trigger [trigger_name]
[before | after]
{insert | update | delete}
on [table_name]
FOR EACH ROW
BEGIN
END;
```

Types

1. DDL Triggers

```
CREATE TRIGGER prevent_table_creation
ON DATABASE
FOR CREATE_TABLE, ALTER_TABLE, DROP_TABLE
AS
BEGIN
  PRINT 'you can not create, drop and alter table in this database';
  ROLLBACK;
END;
```

2. DML Triggers

```
CREATE TRIGGER prevent_update
ON students
FOR UPDATE
AS
BEGIN
  PRINT 'You can not insert, update and delete this table i';
  ROLLBACK;
END;
```

3. Logon Triggers

```
CREATE TRIGGER track_logon
ON LOGON
AS
BEGIN
  PRINT 'A new user has logged in.';
END;
```

Example

```
CREATE TRIGGER stud_marks
BEFORE INSERT ON Student
FOR EACH ROW
SET NEW.total = NEW.subj1 + NEW.subj2 + NEW.subj3,
    NEW.per = (NEW.subj1 + NEW.subj2 + NEW.subj3) * 60 / 100;
```

An **aggregate function** is a function that performs a calculation on a set of values, and returns a single value.
Aggregate functions are often used with the GROUP BY clause of the SELECT statement. The GROUP BY clause splits the result-set into groups of values and the aggregate function can be used to return a single value for each group.
The most commonly used SQL aggregate functions are:

- MIN() - returns the smallest value within the selected column
- MAX() - returns the largest value within the selected column
- COUNT() - returns the number of rows in a set
- SUM() - returns the total sum of a numerical column
- AVG() - returns the average value of a numerical column

Joins

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

Syntax

SELECT column_list

FROM table1

JOIN table2 ON table1.column = table2.column;

Example

Order Table

OrderID	CustomerID	OrderDate
10308	2	1996-09-18
10309	37	1996-09-19
10310	77	1996-09-20

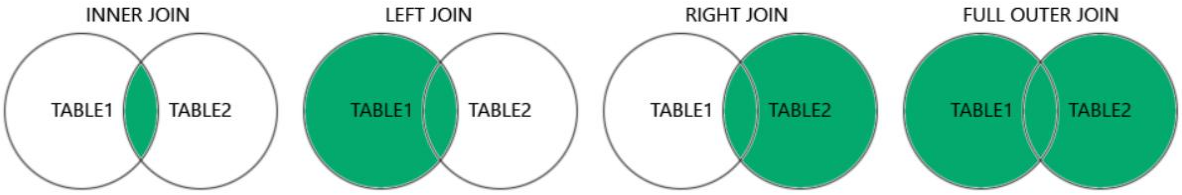
Customers Table

CustomerID	CustomerName	ContactName	Country
1	Alfreds Futterkiste	Maria Anders	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mexico

```
SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate
FROM Orders
INNER JOIN Customers ON Orders.CustomerID=Customers.CustomerID;
```

Types

- (INNER) JOIN: Returns records that have matching values in both tables
- LEFT (OUTER) JOIN: Returns all records from the left table, and the matched records from the right table
- RIGHT (OUTER) JOIN: Returns all records from the right table, and the matched records from the left table
- FULL (OUTER) JOIN: Returns all records when there is a match in either left or right table



Constraints

Constraints can be specified when the table is created with the CREATE TABLE statement, or after the table is created with the ALTER TABLE statement.

It may table or column level

Syntax

```
CREATE TABLE table_name (
   column1 datatype constraint,
   column2 datatype constraint,
   column3 datatype constraint,
   ....
);
```

Types

The following constraints are commonly used in SQL:

- NOT NULL - Ensures that a column cannot have a NULL value
- UNIQUE - Ensures that all values in a column are different
- PRIMARY KEY - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
- FOREIGN KEY - Prevents actions that would destroy links between tables
- CHECK - Ensures that the values in a column satisfies a specific condition
- DEFAULT - Sets a default value for a column if no value is specified

- CREATE INDEX - Used to create and retrieve data from the database very quickly

NOT NULL

```
CREATE TABLE Persons (
  ID int NOT NULL,
  LastName varchar(255) NOT NULL,
  FirstName varchar(255) NOT NULL,
  Age int
);
ALTER TABLE Persons
ALTER COLUMN Age int NOT NULL;
```

UNIQUE

- The UNIQUE constraint ensures that all values in a column are different.
- Both the UNIQUE and PRIMARY KEY constraints provide a guarantee for uniqueness for a column or set of columns.
- A PRIMARY KEY constraint automatically has a UNIQUE constraint.

```
CREATE TABLE Persons (
  ID int NOT NULL UNIQUE,
  LastName varchar(255) NOT NULL,
  FirstName varchar(255),
  Age int
);
```

PRIMARY KEY

```
CREATE TABLE Persons (
  ID int NOT NULL PRIMARY KEY,
  LastName varchar(255) NOT NULL,
  FirstName varchar(255),
  Age int
);
```

FOREIGN KEY

```
CREATE TABLE Orders (
  OrderID int NOT NULL,
  OrderNumber int NOT NULL,
  PersonID int,
  PRIMARY KEY (OrderID),
```

```
    CONSTRAINT FK_PersonOrder FOREIGN KEY (PersonID)
    REFERENCES Persons(PersonID)
);
```

CHECK

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int CHECK (Age>=18)
);

CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    City varchar(255),
    CONSTRAINT CHK_Person CHECK (Age>=18 AND City='Sandnes')
);
```

DEFAULT

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    City varchar(255) DEFAULT 'Sandnes'
);
CREATE TABLE Orders (
    ID int NOT NULL,
    OrderNumber int NOT NULL,
    OrderDate date DEFAULT GETDATE()
);
```

Stored Procedure

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

Syntax

```
CREATE PROCEDURE procedure_name
AS
sql_statement
GO;
EXEC procedure_name;
```

Example

```
CREATE PROCEDURE SelectAllCustomers
AS
SELECT * FROM Customers
GO;
EXEC SelectAllCustomers;
```

Examples of Aggregate Functions

a) SUM() - Total Salary of Employees

sql

CopyEdit

```
SELECT SUM(Salary) AS TotalSalary
FROM Employees;
```

Output:

TotalSalary

5000000

---

b) AVG() - Average Salary of Employees

sql

CopyEdit

```
SELECT AVG(Salary) AS AverageSalary
FROM Employees;
```

Output:

AverageSalary

AverageSalary

50000

---

c) COUNT() - Number of Employees in a Department

sql

CopyEdit

SELECT DepartmentID, COUNT(*) AS EmployeeCount

FROM Employees

GROUP BY DepartmentID;

Output:

DepartmentID EmployeeCount

101 10

102 15

---

d) MAX() and MIN() - Highest and Lowest Salary

sql

CopyEdit

SELECT MAX(Salary) AS HighestSalary, MIN(Salary) AS LowestSalary

FROM Employees;

Output:

HighestSalary LowestSalary

120000 25000

---

3. Using Aggregate Functions with GROUP BY

The **GROUP BY** clause groups data before applying aggregate functions.

Example: Average Salary Per Department

sql

CopyEdit

SELECT DepartmentID, AVG(Salary) AS AvgSalary

FROM Employees

GROUP BY DepartmentID;

Output:

DepartmentID AvgSalary

DepartmentID AvgSalary

101 55000

102 47000

---

4. Using Aggregate Functions with HAVING Clause

The **HAVING** clause filters grouped data after applying an aggregate function.

Example: Departments with More Than 5 Employees

sql

CopyEdit

```
SELECT DepartmentID, COUNT(*) AS EmployeeCount
FROM Employees
GROUP BY DepartmentID
HAVING COUNT(*) > 5;
```

Output:

DepartmentID EmployeeCount

101 10