SQL (Structural Query Language)

Introduction to SQL (Structured Query Language)

SQL (Structured Query Language) is the standard language used to manage and manipulate relational databases. It allows you to perform operations such as creating, reading, updating, and deleting data, as well as managing database structures.

Why Use SQL?

Standardized – Used across many databases (MySQL, SQL Server, PostgreSQL, Oracle, etc.)
Powerful – Handles large amounts of data efficiently
Easy to Learn – Uses simple English-like commands

Data Types in SQL

SQL supports different data types to store and manipulate various kinds of data. These data types vary slightly between database systems (MySQL, SQL Server, PostgreSQL, etc.), but the core concepts remain the same.

1 Numeric Data Types

Used for storing numbers, including integers and decimals.

Data Type	Description	Example
`INT`	Integer values	100, -50
`SMALLINT`	Small-range integer	32767
`BIGINT`	Large-range integer	9223372036854775807
`DECIMAL(p, s)` or `NUMERIC(p, s)`	Fixed-point decimal (p = precision, s = scale)	10.25 (DECIMAL(5,2))
`FLOAT`	Floating-point number	3.14, -45.678
`REAL`	Approximate floating-point number	9.99

2 String (Character) Data Types

Used to store text values.

Data Type	Description	Example
`CHAR(n)`	Fixed-length string (n = length)	'ABC' (CHAR(3))
`VARCHAR(n)`	Variable-length string (max n characters)	'Hello, World!'
`TEXT`	Large text storage (Not indexable)	'This is a long paragraph.'

◇ **Difference Between `CHAR` and `VARCHAR`**

- `CHAR(n)` always reserves `n` characters (e.g., CHAR(10) stores "SQL" as "SQL_____").
- `VARCHAR(n)` only uses space for actual content (e.g., "SQL" in VARCHAR(10) takes 3 bytes).

3 Date & Time Data Types

Used for storing date, time, and timestamps.

Data Type	Description	Example
`DATE`	Stores only the date	'2025-02-18'
`TIME`	Stores only the time	'14:30:00'
`DATETIME`	Stores both date and time	'2025-02-18 14:30:00'
`TIMESTAMP`	Similar to `DATETIME`, but can be auto-updated	'2025-02-18 14:30:00'

4 Boolean Data Type (SQL Server & PostgreSQL)

Stores **TRUE** or **FALSE** values.

Data Type	Description	Example
`BOOLEAN`	Stores TRUE/FALSE values	TRUE, FALSE

◇ **MySQL does not have a `BOOLEAN` type**; instead, it uses `TINYINT(1)` (0 for FALSE, 1 for TRUE).

5 Binary Data Types

Used to store binary data like images, files, or encrypted data.

Data Type	Description	Example
`BLOB`	Binary Large Object	(Stores images, files, etc.)
`VARBINARY(n)`	Variable-length binary data	Binary format of a file

6 Miscellaneous Data Types

Additional data types used in specific databases.

Data Type	Description	Example
`JSON`	Stores JSON objects (MySQL, PostgreSQL)	`{ "name": "John" }`
`UUID`	Universally Unique Identifier	'550e8400-e29b-41d4-a716-446655440000'
`ENUM`	A predefined list of values	ENUM('Male', 'Female', 'Other')

Exercise

1 Create a Table with Different Data Types

```
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY AUTO_INCREMENT,
    Name VARCHAR(100) NOT NULL,
    Email VARCHAR(100) UNIQUE NOT NULL,
    BirthDate DATE,
    RegistrationTime TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    IsActive BOOLEAN DEFAULT TRUE,
    Balance DECIMAL(10,2)
);
```

2 Insert Sample Data

```
INSERT INTO Customers (Name, Email, BirthDate, Balance)
VALUES ('John Doe', 'john@example.com', '1990-05-15', 1200.50);
```

3 Retrieve Data

```
SELECT * FROM Customers;
```

Types of SQL Statements:

1. DDL (Data Definition Language)

Used to define and manage database structures.

- **CREATE** – Creates a new table, view, or database.
- **ALTER** – Modifies an existing table's structure.

- **DROP** – Deletes a table or database.
- **TRUNCATE** – Deletes all records from a table but keeps the structure.

Example:

```
-- Creating a table
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    Name VARCHAR(100),
    Salary DECIMAL(10, 2)
);

-- Modifying a table
ALTER TABLE Employees ADD COLUMN Email VARCHAR(100);

-- Dropping a table
DROP TABLE Employees;
```

Assignment : Employee Management System (DDL & Data Types)

Scenario: A company wants to maintain records of its employees, including their names, departments, salaries, and joining dates.

Task:

1. Create a table `Employees` with appropriate **data types** for:
    - `EmployeeID` (Primary Key)
    - `Name` (Text)
    - `Email` (Unique, Valid Format)
    - `Department` (e.g., HR, IT, Finance)
    - `Salary` (Decimal)
    - `JoiningDate` (Date)
2. Insert at least **5 employees** into the table.
3. Modify the table to **add a column** for `PhoneNumber`.
4. Delete an employee record who **left the company**.
5. Drop the table after verifying all records.

2. DML (Data Manipulation Language)

Used to manipulate data within tables.

- **INSERT** – Adds new records.
- **UPDATE** – Modifies existing records.
- **DELETE** – Removes records.

Example:

```
-- Inserting data
INSERT INTO Employees (EmployeeID, Name, Salary)
```

```
VALUES (1, 'John Doe', 50000);

-- Updating data
UPDATE Employees SET Salary = 55000 WHERE EmployeeID = 1;

-- Deleting data
DELETE FROM Employees WHERE EmployeeID = 1;
```

Assignment : Online Shopping System (DML – Insert, Update, Delete)

Scenario: An online shopping system needs to manage customer orders.

Task:

1. Create a table `Orders` with columns:
    - `OrderID` (Primary Key)
    - `CustomerName`
    - `ProductName`
    - `Quantity`
    - `Price`
    - `OrderDate`
2. Insert at least **10 sample orders**.
3. Update the quantity of a product for a customer.
4. Delete an order where a customer **canceled** the purchase.

3. DQL (Data Query Language)

Used to query and retrieve data from a database.

- **SELECT** – Retrieves data from one or more tables.

Example:

```
-- Selecting all columns
SELECT * FROM Employees;

-- Selecting specific columns
SELECT Name, Salary FROM Employees;

-- Using WHERE clause
SELECT * FROM Employees WHERE Salary > 40000;

-- Using ORDER BY
SELECT * FROM Employees ORDER BY Salary DESC;

-- Using GROUP BY and aggregate functions
SELECT Department, AVG(Salary) AS AvgSalary
FROM Employees
GROUP BY Department;
```

Assignment: Hospital Management System (DQL – Querying the Database)

Scenario: A hospital needs a database to manage **patients and doctors**.

Task:

1. Create a table `Patients` with:
    - `PatientID` (Primary Key)
    - `Name`
    - `Age`
    - `Disease`
    - `DoctorAssigned`
2. Insert at least **7 patient records**.
3. Retrieve the names of **patients above 60 years**.
4. Retrieve a list of **patients assigned to a specific doctor**.
5. Retrieve **count of patients** per disease.

4. DCL (Data Control Language)

Used to control access to data.

- **GRANT** – Gives a user access privileges.
- **REVOKE** – Removes access privileges.

Example:

```
-- Granting privileges
GRANT SELECT, INSERT ON Employees TO 'username';

-- Revoking privileges
REVOKE INSERT ON Employees FROM 'username';
```

`GRANT` and `REVOKE` with a Real-Life Example in SQL Server

Example:

Imagine a **library system** where:

- A **librarian** has full access to manage books.
- A **member** can only read books.

In SQL Server, the **database administrator (DBA)** controls who can access and modify data using **`GRANT` (allow access) and `REVOKE` (remove access).**

SQL Server Example

Let's assume we have a database **LibraryDB** with a table **Books**.

Step 1: Create a User (Library Member)

First, create a login and a database user:

```
CREATE LOGIN LibraryMember WITH PASSWORD = 'Secure@123';

USE LibraryDB;

CREATE USER LibraryMember FOR LOGIN LibraryMember;
```

Step 2: Grant Read-Only Access

Now, allow the `LibraryMember` to **only read (SELECT)** books:

```
GRANT SELECT ON Books TO LibraryMember;


-- Switch to LibraryMember user
EXECUTE AS USER = 'LibraryMember';
```

Now, `LibraryMember` can execute:

```
SELECT * FROM Books;
```

⃠ But cannot INSERT, UPDATE, or DELETE records.

Back to original user

```
REVERT;
```

Step 3: Revoke Read Access

Later, if the library decides to **remove access**, use:

```
REVOKE SELECT ON Books FROM LibraryMember;
```

Now, if `LibraryMember` tries to:

```
SELECT * FROM Books;
```

They will get a **permission denied error**.

Step 4: Grant Full Access to a Librarian

If there's a **librarian** who should manage books, create a user:

```
CREATE LOGIN Librarian WITH PASSWORD = 'Secure@123';

USE LibraryDB;

CREATE USER Librarian FOR LOGIN Librarian;
```

Grant full access to manage books:

```
GRANT SELECT, INSERT, UPDATE, DELETE ON Books TO Librarian;
```

Now, `Librarian` can **view, add, modify, and delete books**.

Step 5: Revoke Write Access

If the librarian should no longer delete books, remove delete permission:

```
REVOKE DELETE ON Books FROM Librarian;
```

Now, `Librarian` can still **read, insert, and update**, but **not delete** books.

Summary

- **`GRANT`** → Gives specific permissions (like reading, writing, modifying data).
- **`REVOKE`** → Removes the granted permissions.
- Used for controlling access in a database, similar to **giving keys and taking them back** in a real-world system.

Setting Roles for a User in SQL Server

In SQL Server, **roles** help manage user permissions efficiently. Instead of granting/revoking permissions one by one, you can assign a **role** to a user.

1. Types of Roles in SQL Server

There are two main types of roles:

1. **Server-Level Roles** (Manage permissions at the SQL Server level)
    - Example: `sysadmin`, `securityadmin`, `dbcreator`
2. **Database-Level Roles** (Manage permissions inside a database)
    - Example: `db_owner`, `db_datareader`, `db_datawriter`

2. Assigning a Role to a User

Step 1: Create a Login (if not already created)

```
CREATE LOGIN user1 WITH PASSWORD = 'Secure@123';
```

Step 2: Create a Database User for the Login

```
USE LibraryDB;  -- Replace with your database name
CREATE USER user1 FOR LOGIN user1;
```

Step 3: Assign a Database Role

Now, assign a role to `user1`.

Option 1: Give Full Access (db_owner)

```
ALTER ROLE db_owner ADD MEMBER user1;
```

Option 2: Give Read-Only Access (db_datareader)

```
ALTER ROLE db_datareader ADD MEMBER user1;
```

Option 3: Give Read & Write Access (db_datareader + db_datawriter)

```
ALTER ROLE db_datareader ADD MEMBER user1;
ALTER ROLE db_datawriter ADD MEMBER user1;
```

3. Verify User Role

To check which roles are assigned to `user1`:

```
SELECT DP1.name AS DatabaseRole, DP2.name AS UserName
FROM sys.database_role_members AS DRM
JOIN sys.database_principals AS DP1 ON DRM.role_principal_id =
DP1.principal_id
JOIN sys.database_principals AS DP2 ON DRM.member_principal_id =
DP2.principal_id
WHERE DP2.name = 'user1';
```

4. Remove a User from a Role

If you need to **remove `user1` from a role**, use:

```
ALTER ROLE db_datareader DROP MEMBER user1;
```

5. Assign a Custom Role (Advanced)

If built-in roles are not enough, you can create a **custom role**:

```
CREATE ROLE LibrarianRole;
GRANT SELECT, INSERT, UPDATE ON Books TO LibrarianRole;
ALTER ROLE LibrarianRole ADD MEMBER user1;
```

Now, `user1` has **read, insert, and update access** to the `Books` table.

Summary

Role Name	Permissions
`db_owner`	Full access
`db_datareader`	Read-only access
`db_datawriter`	Insert & update access
Custom Role	Specific permissions

5. TCL (Transaction Control Language)

Used to manage transactions.

- **COMMIT** – Saves changes made in the transaction.
- **ROLLBACK** – Reverts changes if an error occurs.
- **SAVEPOINT** – Sets a savepoint within a transaction.

Example:

```
-- Starting a transaction
BEGIN TRANSACTION;

-- Inserting data
INSERT INTO Employees (EmployeeID, Name, Salary)
VALUES (2, 'Jane Doe', 60000);

-- Committing the transaction
COMMIT;

-- Rolling back the transaction
ROLLBACK;
```

Example 1: Banking System: Money Transfer (COMMIT & ROLLBACK)

Scenario: A customer transfers ₹5000 from their account to another account. If any error occurs (e.g., insufficient balance), the transaction should be **rolled back**.

Example 2: E-Commerce Order Processing (SAVEPOINT & ROLLBACK TO SAVEPOINT)

Scenario: A customer places an order. First, stock availability is checked, then payment is processed. If payment fails, we **rollback** only the payment step, keeping stock allocation.

Assignment : Movie Ticket Booking System (TCL – COMMIT, ROLLBACK, SAVEPOINT)

Scenario: A **movie ticket booking system** allows users to book tickets. If a user cancels a booking, the system should **rollback** the transaction.

Task:

1. Create a table `Bookings` with:
   - `BookingID` (Primary Key)
   - `CustomerName`
   - `MovieName`
   - `SeatsBooked`
   - `TotalPrice`
2. Insert **4 movie bookings**.
3. Apply `SAVEPOINT` before updating booking details.
4. If a customer cancels, rollback to the **previous savepoint**.
5. Commit the transaction if everything is correct.

Assignment : Bank Transactions & User Permissions (DCL & TCL)

Scenario: A bank wants to manage **customer transactions** while ensuring **data security** using **user roles**.

Task:

1. Create a table `BankAccounts` with columns:
   - `AccountID` (Primary Key)
   - `CustomerName`
   - `AccountType` (Savings/Current)
   - `Balance`
2. Insert **5 customers** with sample balances.
3. Create a transaction for withdrawing ₹5000 from an account.
4. If the balance is **less than ₹5000**, rollback the transaction.
5. Grant **SELECT permissions** to a **customer role** but restrict `UPDATE` permissions.
6. Revoke the **DELETE permission** from users.

Operators

Arithmetic Operators

```
SELECT Name, Salary, Salary + 5000 AS IncreasedSalary FROM Employees;
```

- *Comparison Operators*

```
SELECT * FROM Employees WHERE Age > 30;
```

- *Logical Operators*

```
SELECT * FROM Employees WHERE Age > 25 AND Salary > 60000;
```

- *String Operators*

```
SELECT * FROM Employees WHERE Name LIKE 'J%'; -- Names starting with 'J'
```

- *Conditional (CASE)*

```
SELECT Name, Age,
       CASE
           WHEN Age >= 60 THEN 'Senior'
           WHEN Age >= 18 THEN 'Adult'
           ELSE 'Minor'
       END AS AgeGroup
FROM Employees;
```

- *NULL Handling*

```
SELECT * FROM Employees WHERE Email IS NULL;
```

- **Full Query Example**

```
SELECT Name, Age, Salary, City, Email,
       CASE
           WHEN Age >= 60 THEN 'Senior'
           WHEN Age >= 18 THEN 'Adult'
           ELSE 'Minor'
       END AS AgeGroup
FROM Employees
WHERE (Salary BETWEEN 50000 AND 100000)
AND City IN ('New York', 'Los Angeles')
AND Email IS NOT NULL;
```

GROUP BY

The **`GROUP BY`** clause is used to **group rows that have the same values** in specified columns. It is often used with **aggregate functions** like `COUNT()`, `SUM()`, `AVG()`, `MAX()`, and `MIN()`.

Syntax

```
SELECT column_name, AGGREGATE_FUNCTION(column_name)
FROM table_name
GROUP BY column_name;
```

◇ Example Queries on the `Employees` Table

Let's use the **Employees** table you created earlier.

1 Group Employees by City and Count Them

```
SELECT City, COUNT(*) AS EmployeeCount
FROM Employees
GROUP BY City;
```

◇ Explanation:

- Groups employees by `City`.
- Counts how many employees are in each city.

2 Find Average Salary per City

```
SELECT City, AVG(Salary) AS AvgSalary
FROM Employees
GROUP BY City;
```

◇ Explanation:

- Groups employees by `City`.
- Finds the **average salary** in each city.

3 Find Total Salary per City

```
SELECT City, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY City;
```

◇ Explanation:

- Groups employees by `City`.
- Calculates the **total salary** in each city.

4 Count Employees by Age Group

```
SELECT
    CASE
        WHEN Age >= 60 THEN 'Senior'
        WHEN Age >= 18 THEN 'Adult'
        ELSE 'Minor'
    END AS AgeGroup,
    COUNT(*) AS EmployeeCount
FROM Employees
GROUP BY
    CASE
        WHEN Age >= 60 THEN 'Senior'
        WHEN Age >= 18 THEN 'Adult'
        ELSE 'Minor'
    END;
```

◇ **Explanation:**

- Groups employees based on their age category.
- Counts the number of employees in each age group.

5 Find Maximum Salary in Each City

```
SELECT City, MAX(Salary) AS MaxSalary
FROM Employees
GROUP BY City;
```

◇ **Explanation:**

- Groups employees by `City`.
- Finds the **maximum salary** in each city.

6 Filter Groups Using `HAVING`

If you want to filter grouped results, use `HAVING` instead of `WHERE`.

```
SELECT City, COUNT(*) AS EmployeeCount
FROM Employees
GROUP BY City
HAVING COUNT(*) > 1;
```

◇ **Explanation:**

- Groups employees by `City`.
- Filters only cities where the **employee count is greater than 1**.

◇ Summary

Function	Description
`COUNT(*)`	Counts the number of rows in each group.
`SUM(column_name)`	Calculates the total sum for each group.
`AVG(column_name)`	Finds the average value for each group.
`MAX(column_name)`	Gets the maximum value in each group.
`MIN(column_name)`	Gets the minimum value in each group.
`HAVING`	Filters grouped results (used after `GROUP BY`).

◇ Full Query Example

```
SELECT City, COUNT(*) AS EmployeeCount, AVG(Salary) AS AvgSalary
FROM Employees
GROUP BY City
HAVING AVG(Salary) > 60000;
```

ORDER BY

The **`ORDER BY`** clause is used to **sort** the result set in **ascending (`ASC`)** or **descending (`DESC`)** order based on one or more columns.

Syntax

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

- **`ASC`** → Default sorting (ascending order: lowest to highest).
- **`DESC`** → Sorts in descending order (highest to lowest)

Constraints in SQL Server

Constraints in SQL Server **enforce rules** on data in a table to **maintain accuracy and integrity**.

◇ Types of Constraints

Constraint	Description
PRIMARY KEY	Ensures each row has a **unique** and **non-null** value.
FOREIGN KEY	Enforces a link between two tables.
UNIQUE	Ensures all values in a column are **distinct**.
NOT NULL	Prevents a column from storing `NULL` values.
CHECK	Validates that values meet a specific condition.
DEFAULT	Assigns a default value if no value is provided.

◇ Example Table with Constraints

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,  -- Primary Key
    Name VARCHAR(50) NOT NULL,  -- Cannot be NULL
    Age INT CHECK (Age >= 18),  -- Age must be at least 18
    Salary DECIMAL(10,2) DEFAULT 50000,  -- Default Salary
    DepartmentID INT,
    CONSTRAINT FK_Department FOREIGN KEY (DepartmentID) REFERENCES
Departments(DepartmentID),  -- Foreign Key
    CONSTRAINT UQ_Name UNIQUE (Name)  -- Unique Name
);
```

◇ Explanation

1. **`PRIMARY KEY` (EmployeeID)** → Ensures each employee has a unique ID.
2. **`NOT NULL` (Name)** → Prevents null values.
3. **`CHECK` (Age)** → Ensures age is **18 or older**.
4. **`DEFAULT` (Salary)** → If no salary is provided, it defaults to **50,000**.
5. **`FOREIGN KEY` (DepartmentID)** → Ensures valid department references from **Departments table**.
6. **`UNIQUE` (Name)** → Prevents duplicate names.

Primary Key and Foreign Key in SQL Server

This example demonstrates how to use **Primary Key (PK)** and **Foreign Key (FK)** constraints in SQL Server.

Step 1: Create the Parent Table (`Departments`)

The **`Departments`** table will have a **Primary Key (`DepartmentID`)**.

```
CREATE TABLE Departments (
    DepartmentID INT PRIMARY KEY,   -- Primary Key
    DepartmentName VARCHAR(50) NOT NULL
);
```

Explanation:

- `DepartmentID` is the **Primary Key**, ensuring each department has a **unique** identifier.
- `DepartmentName` cannot be **NULL**.

Step 2: Create the Child Table (`Employees`)

The **`Employees`** table will have a **Foreign Key (`DepartmentID`)** that references `Departments`.

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,   -- Primary Key
    Name VARCHAR(50) NOT NULL,
    Age INT CHECK (Age >= 18),   -- Age must be at least 18
    Salary DECIMAL(10,2) DEFAULT 50000,   -- Default Salary
    DepartmentID INT,
    CONSTRAINT FK_Department FOREIGN KEY (DepartmentID) REFERENCES
Departments(DepartmentID) -- Foreign Key
);
```

Explanation:

- `EmployeeID` is the **Primary Key**, ensuring uniqueness.
- `DepartmentID` is a **Foreign Key**, linking `Employees` to `Departments`.
- The `CHECK (Age >= 18)` ensures employees must be **18 or older**.
- The `DEFAULT (Salary = 50000)` sets a default salary if none is provided.

◇ Adding Constraints After Table Creation

You can add constraints after creating a table using `ALTER TABLE`.

1 Add a Primary Key

```
ALTER TABLE Employees ADD CONSTRAINT PK_Employee PRIMARY KEY (EmployeeID);
```

2 Add a Foreign Key

```
ALTER TABLE Employees ADD CONSTRAINT FK_Department FOREIGN KEY
(DepartmentID) REFERENCES Departments(DepartmentID);
```

3 Add a Unique Constraint

```
ALTER TABLE Employees ADD CONSTRAINT UQ_Name UNIQUE (Name);
```

4 Add a Check Constraint

```
ALTER TABLE Employees ADD CONSTRAINT CHK_Age CHECK (Age >= 18);
```

Removing Constraints

Use `DROP CONSTRAINT` to remove constraints.

1 Drop a Primary Key

```
ALTER TABLE Employees DROP CONSTRAINT PK_Employee;
```

2 Drop a Foreign Key

```
ALTER TABLE Employees DROP CONSTRAINT FK_Department;
```

3 Drop a Unique Constraint

```
ALTER TABLE Employees DROP CONSTRAINT UQ_Name;
```

4 Drop a Check Constraint

```
ALTER TABLE Employees DROP CONSTRAINT CHK_Age;
```

◇ Practical Example: Testing Constraints

◇ Insert Data (Valid)

```
INSERT INTO Employees (EmployeeID, Name, Age, Salary, DepartmentID)
VALUES (1, 'John Doe', 25, 60000, 101);
```

Success!

◇ Insert Data (Invalid)

```
INSERT INTO Employees (EmployeeID, Name, Age, Salary, DepartmentID)
VALUES (2, 'Jane Smith', 17, 75000, 102);
```

Error: CHECK constraint failed (Age must be ≥ 18).

Joins in SQL:

Used to combine rows from two or more tables based on a related column.

- **INNER JOIN:** Returns records with matching values in both tables.
- **LEFT JOIN:** Returns all records from the left table, and matched records from the right table.
- **RIGHT JOIN:** Returns all records from the right table, and matched records from the left table.
- **FULL OUTER JOIN:** Returns all records when there is a match in either left or right table.
- **CROSS JOIN:** Returns the Cartesian product of both tables.

Example:

```
-- INNER JOIN Example
SELECT e.EmployeeID, e.Name, d.DepartmentName
FROM Employees e
INNER JOIN Departments d ON e.DepartmentID = d.DepartmentID;

-- LEFT JOIN Example
SELECT e.EmployeeID, e.Name, d.DepartmentName
FROM Employees e
LEFT JOIN Departments d ON e.DepartmentID = d.DepartmentID;
```

Subqueries and Nested Queries:

Queries within another query.

Example:

```
-- Subquery Example
SELECT Name, Salary
FROM Employees
WHERE Salary > (SELECT AVG(Salary) FROM Employees);
```

Functions in SQL:

1. **Aggregate Functions:**
    - `COUNT()`, `SUM()`, `AVG()`, `MAX()`, `MIN()`
2. **String Functions:**
    - `CONCAT()`, `UPPER()`, `LOWER()`, `SUBSTRING()`

3. **Date Functions:**
    - `CURDATE(),NOW(),DATEDIFF()`

Example:

```
-- Using aggregate function
SELECT COUNT(*) AS TotalEmployees FROM Employees;

-- Using string function
SELECT CONCAT(Name, ' earns ', Salary) AS EmployeeInfo FROM Employees;
```

Constraints in SQL:

- **NOT NULL** – Ensures a column cannot have a NULL value.
- **UNIQUE** – Ensures all values in a column are unique.
- **PRIMARY KEY** – Uniquely identifies each row.
- **FOREIGN KEY** – Maintains referential integrity between tables.
- **CHECK** – Ensures values in a column satisfy a specific condition.
- **DEFAULT** – Assigns a default value to a column if no value is specified.

Example:

```
CREATE TABLE Products (
    ProductID INT PRIMARY KEY,
    ProductName VARCHAR(100) NOT NULL,
    Price DECIMAL(10, 2) CHECK (Price > 0),
    Quantity INT DEFAULT 0
);
```

Indexes in SQL:

Indexes improve the speed of data retrieval.

Example:

```
-- Creating an index
CREATE INDEX idx_name ON Employees (Name);

-- Dropping an index
DROP INDEX idx_name ON Employees;
```

Views in SQL:

A view is a virtual table based on the result set of a SELECT query.

Example:

```
-- Creating a view
CREATE VIEW HighSalaryEmployees AS
```

```
SELECT Name, Salary
FROM Employees
WHERE Salary > 60000;

-- Using the view
SELECT * FROM HighSalaryEmployees;
```

Stored Procedures and Functions:

Reusable SQL code stored in the database.

- **Stored Procedure:** Executes a set of SQL statements.
- **Function:** Returns a single value.

Example:

```
-- Creating a stored procedure
CREATE PROCEDURE GetEmployeeCount()
BEGIN
    SELECT COUNT(*) AS TotalEmployees FROM Employees;
END;

-- Calling the stored procedure
CALL GetEmployeeCount();
```

Assignment: Implementing SQL Constraints in a Library Management System

Scenario:

You are tasked with designing a database for a library management system. The system should manage information about books, authors, members, and book loans. Implement appropriate SQL constraints to ensure data accuracy and consistency.

Requirements:

1. Tables and Constraints:

Authors Table:

AuthorID: Unique identifier for each author.

Constraint: Primary Key (PRIMARY KEY)

FirstName: Author's first name.

Constraint: Cannot be null (NOT NULL)

LastName: Author's last name.

Constraint: Cannot be null (NOT NULL)

Books Table:

BookID: Unique identifier for each book.

Constraint: Primary Key (PRIMARY KEY)

Title: Title of the book.

Constraint: Cannot be null (NOT NULL)

AuthorID: Identifier linking to the author of the book.

Constraint: Foreign Key (FOREIGN KEY) referencing Authors(AuthorID)

PublishedYear: Year the book was published.

Constraint: Check (CHECK) to ensure the year is reasonable (e.g., between 1500 and the current year).

Constraint: Unique (UNIQUE) to prevent duplicate entries.

Members Table:

MemberID: Unique identifier for each member.

Constraint: Primary Key (PRIMARY KEY)

FirstName: Member's first name.

Constraint: Cannot be null (NOT NULL)

LastName: Member's last name.

Constraint: Cannot be null (NOT NULL)

Email: Member's email address.

Constraint: Unique (UNIQUE) and cannot be null (NOT NULL).

Loans Table:

LoanID: Unique identifier for each loan transaction.

Constraint: Primary Key (PRIMARY KEY)

BookID: Identifier of the borrowed book.

Constraint: Foreign Key (FOREIGN KEY) referencing Books(BookID).

MemberID: Identifier of the member who borrowed the book.

Constraint: Foreign Key (FOREIGN KEY) referencing Members(MemberID).

LoanDate: Date when the book was borrowed.

Constraint: Cannot be null (NOT NULL).

ReturnDate: Date when the book was returned.

Constraint: Check (CHECK) to ensure ReturnDate is either null or after LoanDate.