21CSE354T - FULL STACK WEB DEVELOPMENT

Unit-3 - SQL, JDBC Overview

SQL Overview: **Basics of SQL:** Data basics, **Retrieval:** Basic Selection, Joins, **Set Queries:** UNION, INTERSECT, and EXCEPT, Subqueries, **Modifying Data:** Insert, Delete, Update, Creating, Deleting, and Altering Tables,

JDBC: Introduction to JDBC: Setting up the database, **Connecting to a Database:** The Connection Interface, connecting to the database using Driver Manager, Querying and Updating the Database: Statement Interface, Result Set Interface, Querying and Updating the Database.

SQL Overview: Key Concepts and Basics:

What is SQL?

SQL (Structured Query Language) is a standard programming language specifically designed for managing and manipulating relational databases. It allows users to perform various operations like querying data, updating records, and managing database structures.

What is a Database?

- Database: A structured collection of data that is stored and accessed electronically. Databases can be managed by a Database Management System (DBMS).
- Relational Database: A type of database that stores data in tables (relations), where each table consists of rows and columns.

History and Evolution of SQL:

- Origin: SQL was developed in the 1970s by IBM researchers Donald D. Chamberlin and Raymond F. Boyce. It was originally called SEQUEL (Structured English Query Language).
- Standardization: SQL was standardized by the American National Standards Institute (ANSI) in 1986 and the International Organization for Standardization (ISO) in 1987.
- Modern Use: SQL has evolved over time, with various dialects emerging, such as MySQL, PostgreSQL, Microsoft SQL Server, and Oracle SQL.

Key Components of SQL:

- Data Query Language (DQL): Used for querying data from the database.

  - SELECT: Retrieves data from the database.

- Data Definition Language (DDL): Used for defining and managing the structure of the database.
  - CREATE: Creates new database objects like tables, indexes, and views.
  - ALTER: Modifies existing database objects.
  - DROP: Deletes database objects.
- Data Manipulation Language (DML): Used for inserting, updating, and deleting data.
  - INSERT: Adds new data to a table.
  - UPDATE: Modifies existing data in a table.
  - DELETE: Removes data from a table.
- Data Control Language (DCL): Used for controlling access to data in the database.
  - GRANT: Gives a user permission to perform operations on database objects.
  - REVOKE: Removes a user's permissions.
- Transaction Control Language (TCL): Used for managing transactions in a database.
  - COMMIT: Saves changes made in a transaction.
  - ROLLBACK: Reverts changes made in a transaction.
  - SAVEPOINT: Sets a point within a transaction to which you can later roll back.

Relational Database Concepts:

- Database: A collection of organized data that can be easily accessed, managed, and updated.
- Table: The primary structure in a relational database, consisting of rows and columns.
  - Row (Record): A single, horizontal entry in a table that represents a single data item or record.
  - Column (Field): A vertical set of data in a table that represents a specific attribute of the entity (e.g., name, age). Primary Key: A unique identifier for each record in a table.
- Primary Key: A unique identifier for a row in a table. No two rows can have the same primary key value.
- Foreign Key: A field in one table that uniquely identifies a row of another table, creating a relationship between the two tables.

- Index: A database object that improves the speed of data retrieval operations on a table at the cost of additional storage and maintenance.

SQL Basics: (Basic Statements)

- SELECT Statement: Used to query and retrieve data from one or more tables.
  - Basic Syntax:

    SELECT column1, column2 FROM table_name WHERE condition;
- INSERT Statement: Used to add new rows of data to a table.
  - Basic Syntax:

    INSERT INTO table_name (column1, column2) VALUES (value1, value2);
- UPDATE Statement: Used to modify existing records in a table.
  - Basic Syntax:

    UPDATE table_name SET column1 = value1 WHERE condition;
- DELETE Statement: Used to remove records from a table.
  - Basic Syntax:

    DELETE FROM table_name WHERE condition;

Data Manipulation Language (DML):

- Data Manipulation Language (DML): A subset of SQL commands used to add, update, delete, and retrieve data within a database.
  - SELECT: Retrieves data.
  - INSERT INTO: Adds new data.
  - UPDATE: Modifies existing data.
  - DELETE: Removes data.

Data Definition Language (DDL):

- Data Definition Language (DDL): A subset of SQL commands used to define the database schema and structure.
  - CREATE TABLE: Defines a new table and its columns.

    Syntax: CREATE TABLE table_name (column1 datatype, column2 datatype, ...);

- ALTER TABLE: Modifies an existing table, such as adding or removing columns.

  Syntax: ALTER TABLE table_name ADD column_name datatype;

- DROP TABLE: Deletes a table and its data.

  Syntax: DROP TABLE table_name;

SQL Data Types:

SQL Data Types define the kind of data that can be stored in a column of a table in a database. Each column in a database table is assigned a data type, which dictates what kind of values it can hold, such as integers, floating-point numbers, strings, dates, and more.

Common Data Types:

- **Numeric Datatype**
  - **INT:** For integer numbers.
  - **DECIMAL/ NUMERIC:** For fixed-point numbers.
  - **FLOAT / REAL:** floating-point numbers
- **String Data Types**
  - **CHAR:** Stores fixed length strings
  - **VARCHAR:** Stores variable-length strings
  - **TEXT:** Stores large variable-length strings
- **Date and Time Data Types**
  - **DATE:** Stores date values in the format YYYY-MM-DD
  - **TIME:** Stores time values in the format HH:MM:SS.
  - **DATETIME / TIMESTAMP:** Stores both date and time in the format YYYY-MM-DD HH:MM:SS
- **Boolean Data Type**
  - **BOOLEAN/BOOL:** For TRUE or FALSE values.
- **Binary Data Types**
  - **BINARY:** Stores fixed-length binary data
  - **VARBINARY:** Stores variable-length binary data
- **Other Data Types**
  - **ENUM:** A string object that can have one of several predefined values

- **UUID:** Stores a universally unique identifier

Numeric Data Types

- **INT (INTEGER):**
  - Description: Stores whole numbers (both positive and negative).
  - Range: Typically -2,147,483,648 to 2,147,483,647 for a 4-byte INT.
  - Usage: For storing quantities, counters, or any data that doesn't require decimal points.
  - Example:

    CREATE TABLE Employees (EmployeeID INT PRIMARY KEY, Age INT);
- **DECIMAL(p, s) / NUMERIC(p, s):**
  - Description: Stores fixed-point numbers. p denotes precision (total number of digits), and s denotes scale (number of digits after the decimal point).
  - Usage: Ideal for monetary values where precision is crucial, such as prices or financial data.
  - Example:

    CREATE TABLE Products (ProductID INT PRIMARY KEY, Price DECIMAL(10, 2));
- **FLOAT / REAL:**
  - Description: Stores floating-point numbers. FLOAT allows for more precision than REAL.
  - Usage: Suitable for scientific data, measurements, or any data requiring decimal precision.
  - Example:

  CREATE TABLE Measurements (MeasurementID INT PRIMARY KEY, Temperature FLOAT);

String Data Types

- **CHAR(n):**
  - Description: Stores fixed-length strings. n specifies the number of characters.
  - Usage: Best for fields with a known fixed size, like country codes or IDs.
  - Example:

CREATE TABLE Countries (CountryCode CHAR(2) PRIMARY KEY, CountryName VARCHAR(50));

- **VARCHAR(n):**
  - Description: Stores variable-length strings. n specifies the maximum number of characters.
  - Usage: Ideal for names, addresses, or any text that varies in length.
  - Example:

CREATE TABLE Customers (CustomerID INT PRIMARY KEY, FirstName VARCHAR(50), LastName VARCHAR(50));

- **TEXT:**
  - Description: Stores large variable-length strings. Often used for long text fields.
  - Usage: Suitable for storing large bodies of text, such as descriptions, articles, or comments.
  - Example:

CREATE TABLE Articles (ArticleID INT PRIMARY KEY, Cntent TEXT);

Date and Time Data Types

- **DATE:**
  - Description: Stores date values in the format YYYY-MM-DD.
  - Usage: For recording dates without time information, like birthdates or event dates.
  - Example:

CREATE TABLE Events (EventID INT PRIMARY KEY, EventDate DATE);

- **TIME:**
  - Description: Stores time values in the format HH:MM:SS.
  - Usage: For storing times independent of dates, such as shift start times.
  - Example:

CREATE TABLE Schedules (ScheduleID INT PRIMARY KEY, StartTime TIME);

- **DATETIME / TIMESTAMP:**
  - Description: Stores both date and time in the format YYYY-MM-DD HH:MM:SS.

- Usage: For recording exact moments in time, such as order timestamps or log entries.
- Example:

CREATE TABLE Orders (OrderID INT PRIMARY KEY, OrderDate DATETIME);

Boolean Data Type

- **BOOLEAN / BOOL:**
  - Description: Stores TRUE or FALSE values. Depending on the database, it might be stored as 0 (FALSE) and 1 (TRUE).
  - Usage: For fields that need to store binary values, like flags or status indicators.
  - Example:

CREATE TABLE Users (UserID INT PRIMARY KEY, IsActive BOOLEAN);

Binary Data Types

- **BINARY(n):**
  - Description: Stores fixed-length binary data. n specifies the number of bytes.
  - Usage: Suitable for storing binary data like encryption keys or binary hashes.
  - Example:

CREATE TABLE EncryptionKeys (KeyID INT PRIMARY KEY, KeyValue BINARY(16));

- **VARBINARY(n):**
  - Description: Stores variable-length binary data. n specifies the maximum number of bytes.
  - Usage: Useful for storing binary data such as images, files, or multimedia.
  - Example:

CREATE TABLE Files (FileID INT PRIMARY KEY, FileData VARBINARY(8000));

Other Data Types

- **ENUM:**
  - Description: A string object that can have one of several predefined values.
  - Usage: For storing a predefined set of values, like categories or statuses.
  - Example:

CREATE TABLE Employees (EmployeeID INT PRIMARY KEY, EmploymentStatus ENUM('Full-Time', 'Part-Time', 'Contract'));

- **UUID:**
    - Description: Stores a universally unique identifier, often used for primary keys.
    - Usage: Ideal for unique identifiers across different systems, ensuring no collisions.
    - Example:

      CREATE TABLE Devices (DeviceID UUID PRIMARY KEY, DeviceName VARCHAR(50));

Choosing the Right Data Type:

- Consider Storage Requirements: Use data types that match the storage needs of your data (e.g., TINYINT instead of INT for small numbers).
- Performance Implications: Smaller data types can lead to better performance, but ensure they can accommodate your data's range and precision.
- Data Integrity: Choosing the appropriate data type helps maintain data integrity and ensures valid data is stored in the database.

SQL Retrieval: Basic Selection:

SQL retrieval operations are used to fetch data from a database. The most fundamental and frequently used command for data retrieval is the SELECT statement. The SELECT statement allows you to specify the columns you want to retrieve, the tables they come from, and any conditions that must be met.

Basic Syntax of the SELECT Statement

- **Syntax:** SELECT column1, column2, ... FROM table_name;
- **Example:** SELECT FirstName, LastName FROM Employees;

  This query retrieves the FirstName and LastName columns from the Employees table.

Selecting All Columns

- You can select all columns from a table using the asterisk (*) wildcard.
- **Example:** SELECT * FROM Employees;
    - This query retrieves all columns from the Employees table.

Using the WHERE Clause for Filtering

- The WHERE clause is used to filter records based on specific conditions.
- **Syntax:** SELECT column1, column2 FROM table_name WHERE condition;
- **Example:**

  SELECT FirstName, LastName FROM Employees WHERE Department = 'Sales';

  - This query retrieves the first and last names of employees who work in the Sales department.

Common Operators in the WHERE Clause

- **Comparison Operators:** Used to compare values.
  - = (Equal to)
  - != or <> (Not equal to)
  - < (Less than)
  - > (Greater than)
  - <= (Less than or equal to)
  - >= (Greater than or equal to)
- **Example:** SELECT FirstName, LastName FROM Employees WHERE Age > 30;
  - This query selects employees whose age is greater than 30.
- **Logical Operators:** Combine multiple conditions.
  - AND: All conditions must be true.
  - OR: At least one condition must be true.
  - NOT: Negates a condition.
- **Example:**

  SELECT FirstName, LastName FROM Employees

  WHERE Department = 'Sales' AND Age > 30;

  - This query retrieves employees in the Sales department who are older than 30.

Using ORDER BY for Sorting Results

- The ORDER BY clause is used to sort the result set by one or more columns, either in ascending (ASC) or descending (DESC) order.
- **Syntax:** SELECT column1, column2 FROM table_name ORDER BY column1 ASC|DESC;

- **Example:**

  SELECT FirstName, LastName FROM Employees ORDER BY LastName ASC;

  - This query sorts the employees by their last names in ascending order.

Using LIMIT or TOP to Restrict the Number of Results

- **LIMIT:** Used in MySQL, PostgreSQL, and other databases to limit the number of rows returned.
  - **Syntax:** SELECT column1, column2 FROM table_name LIMIT number;
  - **Example:** SELECT FirstName, LastName FROM Employees LIMIT 5;
    - This query retrieves the first 5 employees from the Employees table.
- **TOP:** Used in SQL Server to limit the number of rows returned.
  - **Syntax:** SELECT TOP number column1, column2 FROM table_name;
  - **Example:** SELECT TOP 5 FirstName, LastName FROM Employees;
    - This query retrieves the top 5 employees from the Employees table.

Using DISTINCT to Eliminate Duplicate Records

- The DISTINCT keyword is used to return only distinct (different) values, eliminating duplicates from the result set.
- **Syntax:** SELECT DISTINCT column1, column2 FROM table_name;
- **Example:** SELECT DISTINCT Department FROM Employees;
  - This query retrieves a list of unique departments from the Employees table.

Using ALIAS for Renaming Columns or Tables

- An alias is a temporary name given to a column or table in a query. It is often used to make the result set more readable.
- **Syntax:** SELECT column_name AS alias_name FROM table_name AS alias_name;
- **Example:** SELECT FirstName AS First_Name, LastName AS Last_Name

  FROM Employees AS Emp;
  - This query renames the FirstName and LastName columns as First_Name and Last_Name, respectively, and refers to the Employees table as Emp.

Combining Multiple Conditions with IN and BETWEEN

- **IN:** Allows you to specify multiple values in a WHERE clause.
  - **Example:** SELECT FirstName, LastName FROM Employees

WHERE Department IN ('Sales', 'Marketing');

  - This query retrieves employees who work in either the Sales or Marketing department.

- **BETWEEN:** Filters the result set within a specific range.
  - **Example:** SELECT FirstName, LastName FROM Employees

    WHERE Age BETWEEN 30 AND 40;

    - This query selects employees whose age is between 30 and 40.

SQL Joins and Their Types:

SQL Join operation combines data or rows from two or more tables based on a common field between them.

SQL JOIN

SQL JOIN clause is used to query and access data from multiple tables by establishing logical relationships between them. It can access data from multiple tables simultaneously using common key values shared across different tables.

We can use SQL JOIN with multiple tables. It can also be paired with other clauses, the most popular use will be using JOIN with **WHERE clause** to filter data retrieval.

SQL JOIN Example

Consider the two tables below as follows:

Student:

ROLL_NO	NAME	ADDRESS	PHONE	Age
1	HARSH	DELHI	XXXXXXXXXX	18
2	PRATIK	BIHAR	XXXXXXXXXX	19
3	RIYANKA	SILIGURI	XXXXXXXXXX	20
4	DEEP	RAMNAGAR	XXXXXXXXXX	18
5	SAPTARHI	KOLKATA	XXXXXXXXXX	19
6	DHANRAJ	BARABAJAR	XXXXXXXXXX	20
7	ROHIT	BALURGHAT	XXXXXXXXXX	18
8	NIRAJ	ALIPUR	XXXXXXXXXX	19

StudentCourse:

COURSE_ID	ROLL_NO
1	1
2	2
2	3
3	4
1	5
4	9
5	10
4	11

Both these tables are connected by one common key (column) i.e **ROLL_NO.** We can perform a JOIN operation using the given SQL query:

SELECT s.roll_no, s.name, s.address, s.phone, s.age, sc.course_id

FROM Student s

JOIN StudentCourse sc **ON** s.roll_no = sc.roll_no;

Output:

ROLL_NO	NAME	ADDRESS	PHONE	AGE	COURSE_ID
1	HARSH	DELHI	XXXXXXXXXX	18	1
2	PRATIK	BIHAR	XXXXXXXXXX	19	2
3	RIYANKA	SILGURI	XXXXXXXXXX	20	2
4	DEEP	RAMNAGAR	XXXXXXXXXX	18	3
5	SAPTARHI	KOLKATA	XXXXXXXXXX	19	1

Types of JOIN in SQL

Different type of SQL JOIN's are:

- INNER JOIN
- LEFT JOIN
- RIGHT JOIN
- FULL JOIN

- Natural join

SQL INNER JOIN

The **INNER JOIN** keyword selects all rows from both the tables as long as the condition is satisfied. This keyword will create the result-set by combining all rows from both the tables where the condition satisfies i.e value of the common field will be the same.

Syntax:

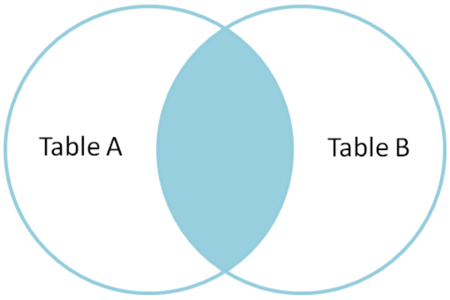
SELECT table1.column1,table1.column2,table2.column1,....

FROM table1

INNER JOIN table2 **ON** table1.matching_column = table2.matching_column;

Here,

- **table1**: First table.
- **table2**: Second table
- **matching_column**: Column common to both the tables.



Example: This query will show the names and age of students enrolled in different courses.

SELECT StudentCourse.COURSE_ID, Student.NAME, Student.AGE **FROM** Student

INNER JOIN

StudentCourse **ON** Student.ROLL_NO = StudentCourse.ROLL_NO;

Output:

COURSE_ID	NAME	Age
1	HARSH	18
2	PRATIK	19
2	RIYANKA	20
3	DEEP	18
1	SAPTARHI	19

SQL LEFT JOIN (LEFT OUTER JOIN)

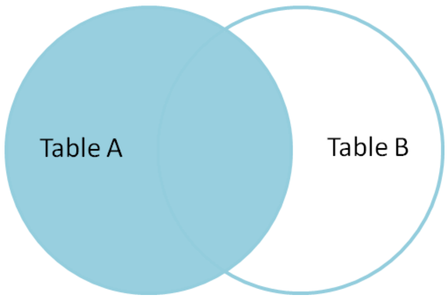
LEFT JOIN returns all the rows of the table on the left side of the join and matches rows for the table on the right side of the join. For the rows for which there is no matching row on the right side, the result-set will contain *null*. LEFT JOIN is also known as LEFT OUTER JOIN.

Syntax:

SELECT table1.column1,table1.column2,table2.column1,.... **FROM** table1

LEFT JOIN

table2 **ON** table1.matching_column = table2.matching_column;



Example:

SELECT Student.NAME,StudentCourse.COURSE_ID **FROM** Student

LEFT JOIN

StudentCourse **ON** StudentCourse.ROLL_NO = Student.ROLL_NO;

Output:

NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
DHANRAJ	*NULL*
ROHIT	*NULL*
NIRAJ	*NULL*

SQL RIGHT JOIN

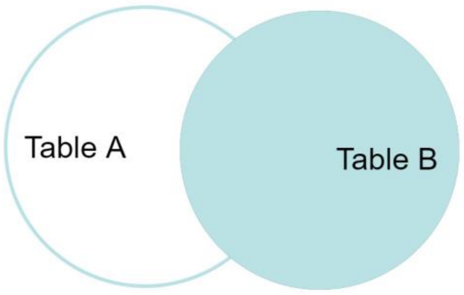
RIGHT JOIN returns all the rows of the table on the right side of the join and matching rows for the table on the left side of the join. It is very similar to LEFT JOIN For the rows for which there is no matching row on the left side, the result-set will contain *null*. RIGHT JOIN is also known as RIGHT OUTER JOIN.

Syntax:

SELECT table1.column1, table1.column2, table2.column1,.... **FROM** table1

RIGHT JOIN

table2 **ON** table1.matching_column = table2.matching_column;



Example:

SELECT Student.NAME,StudentCourse.COURSE_ID **FROM** Student

RIGHT JOIN

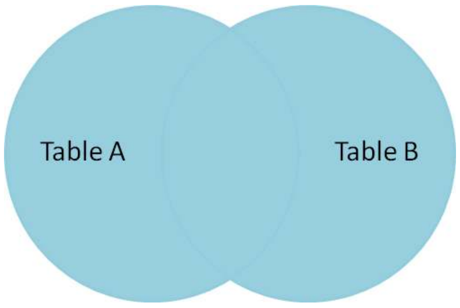
StudentCourse **ON** StudentCourse.ROLL_NO = Student.ROLL_NO;

Output:

NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
NULL	4
NULL	5
NULL	4

SQL FULL JOIN

FULL JOIN creates the result-set by combining results of both LEFT JOIN and RIGHT JOIN. The result-set will contain all the rows from both tables. For the rows for which there is no matching, the result-set will contain *NULL* values.



Syntax

SELECT table1.column1,table1.column2,table2.column1,.... **FROM** table1

FULL JOIN

table2 **ON** table1.matching_column = table2.matching_column;

Example

SELECT Student.NAME,StudentCourse.COURSE_ID **FROM** Student

FULL JOIN

StudentCourse **ON** StudentCourse.ROLL_NO = Student.ROLL_NO;

Output:

NAME	COURSE_ID
HARSH	1
PRATIK	2
RIYANKA	2
DEEP	3
SAPTARHI	1
DHANRAJ	NULL
ROHIT	NULL
NIRAJ	NULL
NULL	4

NAME	COURSE_ID
NULL	5
NULL	4

SQL Natural join (?)

Natural join can join tables based on the common columns in the tables being joined. A natural join returns all rows by matching values in common columns having same name and data type of columns and that column should be present in both tables. Both table must have at least one common column with same column name and same data type.

The two table are joined using **Cross join**.

DBMS will look for a common column with same name and data type Tuples having exactly same values in common columns are kept in result.

Natural join Example:

Look at the two tables below- Employee and Department

Employee		
EmpID	**EmpName**	**DeptID**
1	Ram	10
2	Jon	30
3	Bob	50

Department	
DeptID	**DeptName**
10	IT
30	HR
40	TIS

Problem: Find all Employees and their respective departments.

Solution Query: (Employee) ? (Department)

EmpID	**EmpName**	**DeptID**	**DeptId**	**DeptName**
1	Ram	10	10	IT
2	Jon	30	30	HR
Employee data			Department data	

CROSS JOIN

- **Description:** Returns the Cartesian product of the two tables, pairing each row from the first table with every row from the second table.
- **Syntax:** SELECT columns FROM table1 CROSS JOIN table2;
- **Example:**

SELECT Employee.EmpName, Department.DeptName FROM Employee

CROSS JOIN Departments;

  - **Explanation:** This query retrieves every possible combination of employees and departments. If there are 10 employees and 5 departments, the result will have 50 rows.

SELF JOIN

- **Description:** A self-join is used to join a table with itself, often to compare rows within the same table or to handle hierarchical data.
- **Syntax:** SELECT a.column_name, b.column_name FROM table_name a, table_name b

  WHERE condition;
- **Example:**

SELECT a.EmpID AS Employees, b.EmpID AS Manager

FROM Employee a INNER JOIN Employee b ON a.ManagerID = b.EmployeeID;

  - **Explanation:** This query retrieves employees along with their managers by joining the Employees table with itself. The alias a refers to the employee, and b refers to the manager.

Using Joins with Multiple Tables

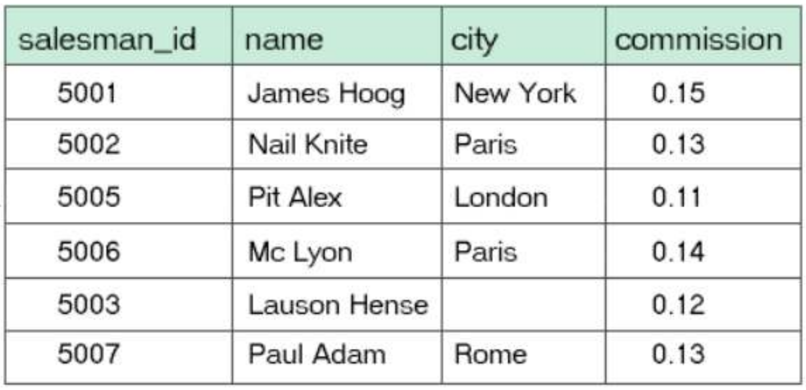
- You can join more than two tables by chaining multiple joins together.
- **Example:**

```
SELECT Employee.EmpName, Department.DeptName.LocationName

FROM Employee

INNER JOIN Department ON Employee.DeptID = Department.DeptID

INNER JOIN Locations ON Department.LocationID = Locations.LocationID;
```

  - **Explanation:** This query retrieves the first and last names of employees, their department names, and the locations of those departments by joining three tables: Employees, Departments, and Locations

Example: 1

1. Write a SQL statement that displays all the information about all salespeople.



salesman_id	name	city	commission
5001	James Hoog	New York	0.15
5002	Nail Knite	Paris	0.13
5005	Pit Alex	London	0.11
5006	Mc Lyon	Paris	0.14
5003	Lauson Hense		0.12
5007	Paul Adam	Rome	0.13

Query: SELECT * FROM salesman;

2. Write a SQL statement to display specific columns such as names and commissions for all salespeople.

Query: SELECT name, commission FROM salesman;

3. Write a query to display a string "Welcome to SQL".

Query: SELECT ‘Welcome to SQL’;

4. From the following table, write a SQL query to locate salespeople who live in the city of 'Paris'. Return salesperson's name, city.

Query: SELECT name,city FROM salesman WHERE city='Paris';

Example: 2

Consider the below two tables for reference while trying to solve the SQL queries for practice.

Table – EmployeeDetails

EmpId	FullName	ManagerId	DateOfJoining	City
121	John Snow	321	01/31/2019	Toronto
321	Walter White	986	01/30/2020	California
421	Kuldeep Rana	876	27/11/2021	New Delhi

Table – EmployeeSalary

EmpId	Project	Salary	Variable
121	P1	8000	500
321	P2	10000	1000
421	P1	12000	0

1. Write an SQL query to fetch the EmpId and FullName of all the employees working under the Manager with id – '986'.

SELECT EmpId, FullName FROM EmployeeDetails WHERE ManagerId = 986;

2. Write an SQL query to fetch the different projects available from the EmployeeSalary table.

SELECT DISTINCT(Project) FROM EmployeeSalary;

3. Write an SQL query to fetch the count of employees working in project 'P1'.
SELECT COUNT(*) FROM EmployeeSalary WHERE Project = 'P1';

4. Write an SQL query to find the maximum, minimum, and average salary of the employees.
SELECT Max(Salary), Min(Salary), AVG(Salary) FROM EmployeeSalary;

5. Write an SQL query to find the employee id whose salary lies in the range of 9000 and 15000.

SELECT EmpId, Salary FROM EmployeeSalary WHERE Salary BETWEEN 9000 AND 15000;

6. Write an SQL query to fetch those employees who live in Toronto and work under the manager with ManagerId – 321.

SELECT EmpId, City, ManagerId FROM EmployeeDetails WHERE City='Toronto' AND ManagerId='321';

7. Write an SQL query to fetch all the employees who either live in California or work under a manager with ManagerId – 321.

SELECT EmpId, City, ManagerId FROM EmployeeDetails WHERE City='California' OR ManagerId='321';

8. Write an SQL query to fetch all those employees who work on Projects other than P1.
SELECT EmpId FROM EmployeeSalary WHERE NOT Project='P1';

(or)

SELECT EmpId FROM EmployeeSalary WHERE Project <> 'P1';

9. Write an SQL query to fetch all the EmpIds which are present in either of the tables – 'EmployeeDetails' and 'EmployeeSalary'.

SELECT EmpId FROM EmployeeDetails UNION SELECT EmpId FROM EmployeeSalary;

10. Write an SQL query to fetch common records between two tables.

SELECT * FROM EmployeeSalary INTERSECT SELECT * FROM ManagerSalary;

11. Write an SQL query to fetch records that are present in one table but not in another table.

SELECT * FROM EmployeeSalary MINUS SELECT * FROM ManagerSalary;

12. Write an SQL query to fetch employee's full names and replace the space with '-'.

SELECT REPLACE(FullName, ' ', '-') FROM EmployeeDetails;

13. Write an SQL query to find the current date-time.

SELECT NOW();

14. Write an SQL query to fetch top n records.

SELECT TOP N * FROM EmployeeSalary ORDER BY Salary DESC;

15. Write an SQL query to fetch all the Employee details from the **EmployeeDetails table who joined in the Year 2020.**

SELECT * FROM EmployeeDetails WHERE DateOfJoining BETWEEN '2020/01/01' AND '2020/12/31';

16. Fetch all the employees who are not working on any project.

SELECT EmpId FROM EmployeeSalary WHERE Project IS NULL;

INTRODUCTION TO JDBC (JAVA DATABASE CONNECTIVITY)

JDBC stands for **Java Database Connectivity**. It is an API (Application Programming Interface) that allows Java applications to interact with databases. Here are some key points about JDBC:

Purpose of JDBC

- **Database Interaction**: JDBC enables Java applications to execute SQL statements, retrieve results, and propagate changes back to the database.
- **Standardization**: It provides a standard interface for connecting to different databases, making Java applications database-independent.

Components of JDBC

1. **JDBC API**: Provides methods and interfaces for database communication.
   - **java.sql**: Contains core interfaces and classes for JDBC.

   - **javax.sql**: Extends java.sql with additional features like connection pooling.
2. **JDBC Driver Manager**: Manages a list of database drivers to establish a connection.
3. **JDBC Test Suite**: Tests the operations performed by JDBC drivers.
4. **JDBC-ODBC Bridge**: Allows JDBC to connect to ODBC databases.

Types of JDBC Drivers

1. **Type 1**: JDBC-ODBC bridge driver.
2. **Type 2**: Native-API driver, uses client-side libraries of the database.
3. **Type 3**: Network protocol driver, uses middleware to convert JDBC calls.
4. **Type 4**: Thin driver, converts JDBC calls directly into database-specific calls.

Setting Up the Database in JDBC

1. **Load the Driver**: Register the driver class.
   ```
   Class.forName("com.mysql.cj.jdbc.Driver");
   ```
2. **Establish a Connection**: Use DriverManager to connect to the database.
   ```
   Connection con = DriverManager.getConnection ("jdbc:mysql: //localhost:3306/
                                                  myDb", "user", "password");
   ```
3. **Create a Statement**: Use the Connection object to create a Statement.
   ```
   Statement stmt = con.createStatement( );
   ```
4. **Execute a Query**: Execute SQL queries using the Statement object.
   ```
   ResultSet rs = stmt.executeQuery("SELECT * FROM myTable");
   ```
5. **Process the Results**: Iterate through the ResultSet to process the results.
   ```
   while (rs.next( ))
   {
           System.out.println(rs.getString(1));
   }
   ```
6. **Close the Connection**: Close the Connection, Statement, and ResultSet objects to free resources.
   ```
           rs.close( );     stmt.close( );          con.close( );
   ```

Example Code

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
public class JDBCDemo
{
   public static void main(String[ ] args)
  {
      try  {
         Class.forName("com.mysql.cj.jdbc.Driver");  // Load the JDBC driver
         // Establish a connection
         Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/myDb",
                                                                       "user", "password");
         Statement stmt = con.createStatement( ); // Create a statement
         ResultSet rs = stmt.executeQuery("SELECT * FROM myTable");  // Execute a query
         while (rs.next( ))   {   // Process the results
                 System.out.println(rs.getString(1));
         }
         // Close the resources
         rs.close( );
         stmt.close( );
         con.close( );
      }
      catch (Exception e) {
         e.printStackTrace( );
      }
   }
}
```

CONNECTING TO THE DATABASE USING DRIVER MANAGER

The Steps to be followed,

1. Install the Database Management System (DBMS)
2. Install the JDBC Driver
3. Establish a Connection
4. Create and Execute SQL Statements
5. Process the Results
6. Close the Connection

1. Install the Database Management System (DBMS)

- **Download and Install**: Install the DBMS of your choice (e.g., MySQL, PostgreSQL, Oracle).
- **Configure the DBMS**: Follow the installation instructions to set up the database server and create a new database.

2. Install the JDBC Driver

- **Download the Driver**: Obtain the JDBC driver for your DBMS from the vendor's website.
- **Add to Project**: Include the JDBC driver JAR file in your project's classpath.

3. Establish a Connection

i. **Import JDBC Packages**:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
```

ii. **Load and Register the Driver**:

```
try
{
        Class.forName("com.mysql.cj.jdbc.Driver"); // For MySQL
}
catch (Exception e)
{
        e.printStackTrace( );
}
```

iii. **Create a Connection using Connection Interface**:

The Connection Interface in JDBC

The Connection interface is a crucial part of the JDBC API. It represents a single database connection and provides various methods to execute SQL statements, manage transactions, and retrieve metadata about the database. Here are some key points about the Connection interface:

Key Features of the Connection Interface

- **Executing SQL Statements**:
  - The Connection interface provides methods to create Statement, PreparedStatement, and CallableStatement objects for executing SQL queries.
  - Statement stmt = con.createStatement( );
  - PreparedStatement pstmt = con.prepareStatement("SELECT * FROM users ");
  - CallableStatement cstmt = con.prepareCall("{call myStoredProcedure( )}");
- **Transaction Management**:
  - It allows you to manage transactions by committing or rolling back changes.
  - con.setAutoCommit(false); // Disable auto-commit mode
  - con.commit( ); // Commit changes
  - con.rollback( ); // Rollback changes if needed
- **Metadata Retrieval**:
  - You can retrieve metadata about the database and the results of SQL queries.
    - DatabaseMetaData dbMetaData = con.getMetaData( );
    - ResultSet rs = dbMetaData.getTables(null, null, "%", new String[ ] {"TABLE"});
- **Connection Pooling**:
  - The Connection interface supports connection pooling, which can improve the performance of database applications by reusing connections.
    - DataSource ds = new MysqlDataSource( );
    - Connection con = ds.getConnection( );
- **Auto-commit Mode**:
  - It supports setting the auto-commit mode for the connection, which determines whether changes are committed automatically after each SQL statement.

con.setAutoCommit(true); // Enable auto-commit mode

4. Create and Execute SQL Statements

- **Create a Statement**:
  - Statement stmt = con.createStatement( );
- **Execute SQL Queries**:
  - String createTableSQL = "CREATE TABLE IF NOT EXISTS users ("name VARCHAR(100), "+ "email VARCHAR(100))";
  - stmt.execute(createTableSQL);
- **Insert Data**:

  String insertSQL = "INSERT INTO users (name, email) VALUES ('John Doe',
  'john@example.com')";

  stmt.executeUpdate(insertSQL);

5. Process the Results

- **Execute a Query**:
  - String selectSQL = "SELECT * FROM users";
  - ResultSet rs = stmt.executeQuery(selectSQL);
- **Iterate Through the Results using ResultSet Interface**:

ResultSet Interface in JDBC

The ResultSet interface in JDBC represents the result set of a database query. It provides methods to iterate through the results, retrieve individual columns by name or index, and update the data. Here are some key points about the ResultSet interface:

Key Features of the ResultSet Interface

- **Cursor Management**:
  - The ResultSet maintains a cursor pointing to its current row of data. Initially, the cursor is positioned before the first row.
  - You can move the cursor using methods like next( ), previous( ), first( ), last( ), and absolute(int row).
  - while (rs.next( ))
  - {
  - // Process each row
  - }
- **Retrieving Data**:
  - The ResultSet interface provides getter methods to retrieve column values from the current row. You can use either the column index or the column name.
    - int id = rs.getInt("id");

- String name = rs.getString(2); // Assuming 'name' is the second column

- **Updating Data**:
  - If the ResultSet is updatable, you can use updater methods to modify the data in the current row.
    - rs.updateString("name", "New Name");
    - rs.updateRow( );
- **Types of ResultSet**:
  - **Forward-only**: The cursor can only move forward.
  - **Scroll-insensitive**: The cursor can move both forward and backward, but changes made by others are not visible.
  - **Scroll-sensitive**: The cursor can move both forward and backward, and changes made by others are visible.
    - Statement stmt = con.createStatement (ResultSet. TYPE_SCROLL_INSENSITIVE,ResultSet. CONCUR_UPDATABLE);
    - ResultSet rs = stmt.executeQuery("SELECT * FROM users");
- **Metadata Retrieval**:
  - You can retrieve metadata about the columns in the ResultSet using the ResultSetMetaData interface.

```
ResultSetMetaData metaData = rs.getMetaData( );
int columnCount = metaData.getColumnCount( );
for (int i = 1; i <= columnCount; i++)
{
System.out.println("Column " + i + ": " + metaData.getColumnName(i));
}
```

6. Close the Connection

Close Resources:

```
rs.close( );

stmt.close( );

con.close( );
```

Example Code

```
import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;
```

```
public class JDBCDemo
{
   public static void main(String[ ] args)
   {
      String url = "jdbc:mysql://localhost:3306/myDb";
      String user = "username";
      String password = "password";
      try
      {
         // Load and register the driver
         Class.forName("com.mysql.cj.jdbc.Driver");
         // Establish a connection
         Connection con = DriverManager.getConnection(url, user, password);
         System.out.println("Connection established successfully!");
         // Create a statement
         Statement stmt = con.createStatement( );
         // Create a table
         String createTableSQL = "CREATE TABLE users ("name VARCHAR(100), "+ "email
VARCHAR(100))";
         stmt.execute(createTableSQL);
         // Insert data
         String insertSQL = "INSERT INTO users (name, email) VALUES ('John Doe',
'john@example.com')";
         stmt.executeUpdate(insertSQL);
         // Execute a query
         String selectSQL = "SELECT * FROM users";
         ResultSet rs = stmt.executeQuery(selectSQL);
         // Process the results
         while (rs.next( ))
        {
           String name = rs.getString("name");
           String email = rs.getString("email");
           System.out.println("Name: " + name + ", Email: " + email);
         }
         // Close the resources
         rs.close( );
         stmt.close( );
         con.close( );
      }
      catch (Exception e)
      {
         e.printStackTrace( );
      }
   }
}
```