JDBC

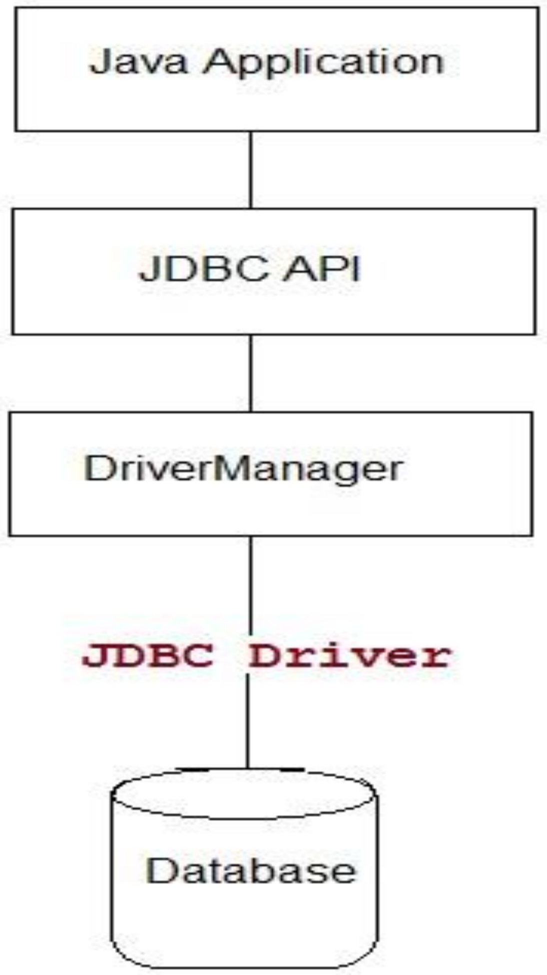
Introduction

Java Database Connectivity (JDBC) is an application programming interface (API) that helps Java program to communicate with databases and manipulates their data. The JDBC API provides the methods that can be used to send SQL and PL/SQL statements to almost any relational database.

Purpose of JDBC API

- To access tables and its data from relation database.
- To send queries and update statement to database.
- Obtain and modify the results to and from a JDBC application.
- Find the metadata of the table.
- Performing different operations on a database, like creating table, querying data, updating data, inserting data from a Java application.

Jdbc Design


Java Application
JDBC API
DriverManager
JDBC Driver
Database

- **Application:** It is a java application, java applet or a servlet which communicates with a data source.
- **The JDBC API:** The JDBC API allows Java programs to execute SQL statements and retrieve results. Some of the important classes and interfaces defined in JDBC API are as follows:
- DriverManager
  Driver
  Connection
  Statement
  PreparedStatement
  CallableStatement
  ResultSet
  SQL data
- **DriverManager:** It plays an important role in the JDBC architecture.It uses some database-specific drivers to effectively connect enterprise applications to databases.
- **JDBC drivers:** To communicate with a data source through JDBC, you need a JDBC driver that intelligently communicates with the respective data source.

JDBC architecture

- The JDBC architecture consists of two-tier and three-tier processing models to access a database. They are as described below:
- **Two-tier model:** A java application communicates directly to the data source. The JDBC driver enables the communication between the application and the data source. When a user sends a query to the data source, the answers for those queries are sent back to the user in the form of results.
  The data source can be located on a different machine on a network to which a user is connected. This is known as a **client/server configuration**, where the user's machine acts as a client and the machine having the data source running acts as the server.

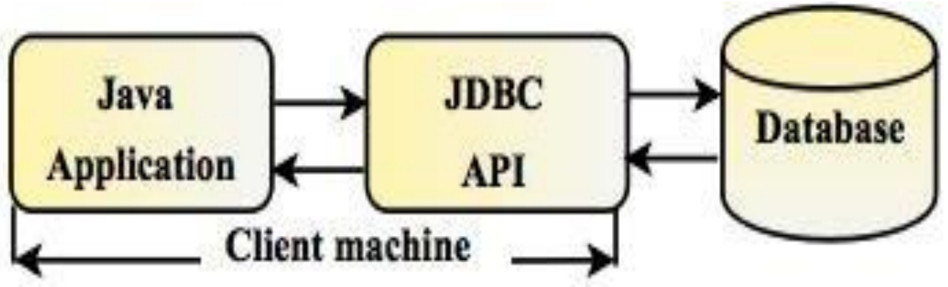


Fig: Two-tier Architecture of JDBC

Three-tier Architecture

In this, the user’s queries are sent to middle-tier services, from which the commands are again sent to the data source. The results are sent back to the middle tier, and from there to the user.
This type of model is found very useful by management information system directors.

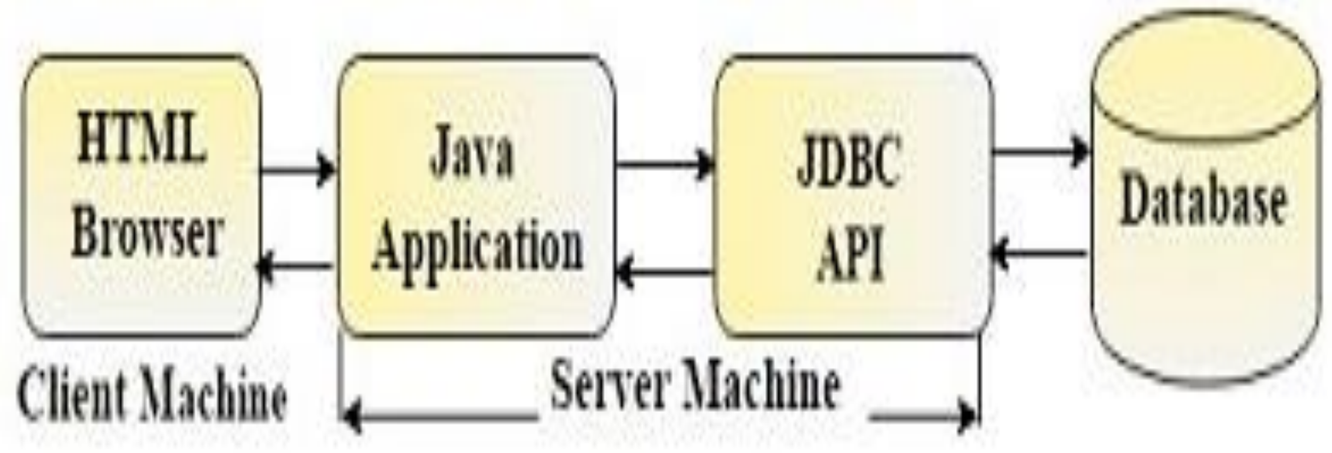


Fig: Three-tier Architecture of JDBC

JDBC Driver

- JDBC Driver is a software component that enables java application to interact with the database. The driver is an interface that existed between Java application and database to map Java API calls to Query language API calls and Query language API calls to Java API calls.
- There are 4 types of JDBC drivers:

1. JDBC-ODBC bridge ODBC driver
2. Native-API partly driver (partially java driver)
3. JDBC-Net pure Java(fully java driver)
4. Native protocol java Driver(Thin driver (fully java driver))

1) TYPE1: JDBC-ODBC bridge ODBC driver

The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. The ODBC driver needs to be installed on the client machine.

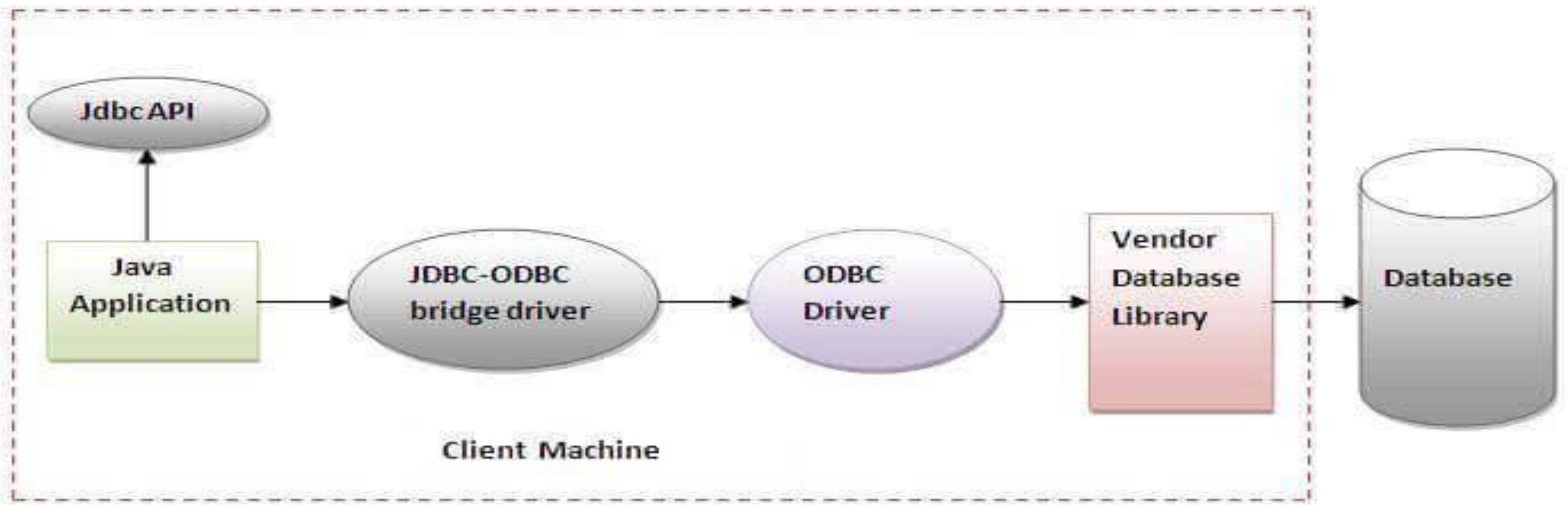


Figure- JDBC-ODBC Bridge Driver

Advantages & Disadvantages

Advantages:

1. easy to use.
2. can be easily connected to any database.

Disadvantages:

1. Performance degraded because JDBC method call is converted into the ODBC function calls.
2. The ODBC driver needs to be installed on the client machine.

Type2: Native-API partly driver (partially java driver)

- The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java. Partially written in java and partially written in c/c++.
- Vendor specific libraries needs to be installed at client machine.

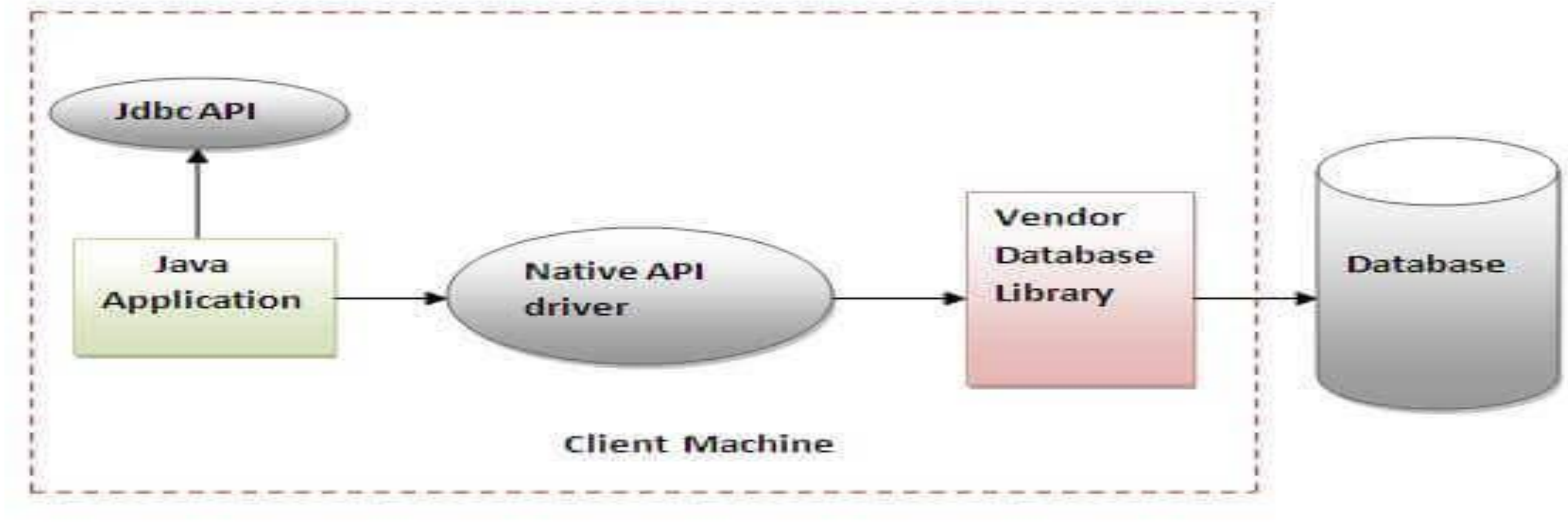


Figure- Native API Driver

Advantages & Disadvantages

Advantage:

1. Faster than JDBC-ODBC bridge driver.

Disadvantage:

1. The vendor client library needs to be installed on the client machine.
2. Not all databases have a client-side library.
3. This driver is platform dependent.
4. This driver supports all Java applications except applets.

Type3: JDBC-Net pure Java(fully java driver)

- The JDBC type 3 driver, also known as the Pure Java driver. It makes use of a middle tier between the calling program and the database. The middle-tier (application server) converts JDBC calls directly or indirectly into a vendor-specific database protocol.
- The same client-side JDBC driver may be used for multiple databases. It depends on the number of databases the middleware has been configured to support. The type 3 driver is platform-independent as the platform-related differences are taken care of by the middleware. Also, making use of the middleware provides additional advantages of security and firewall access.

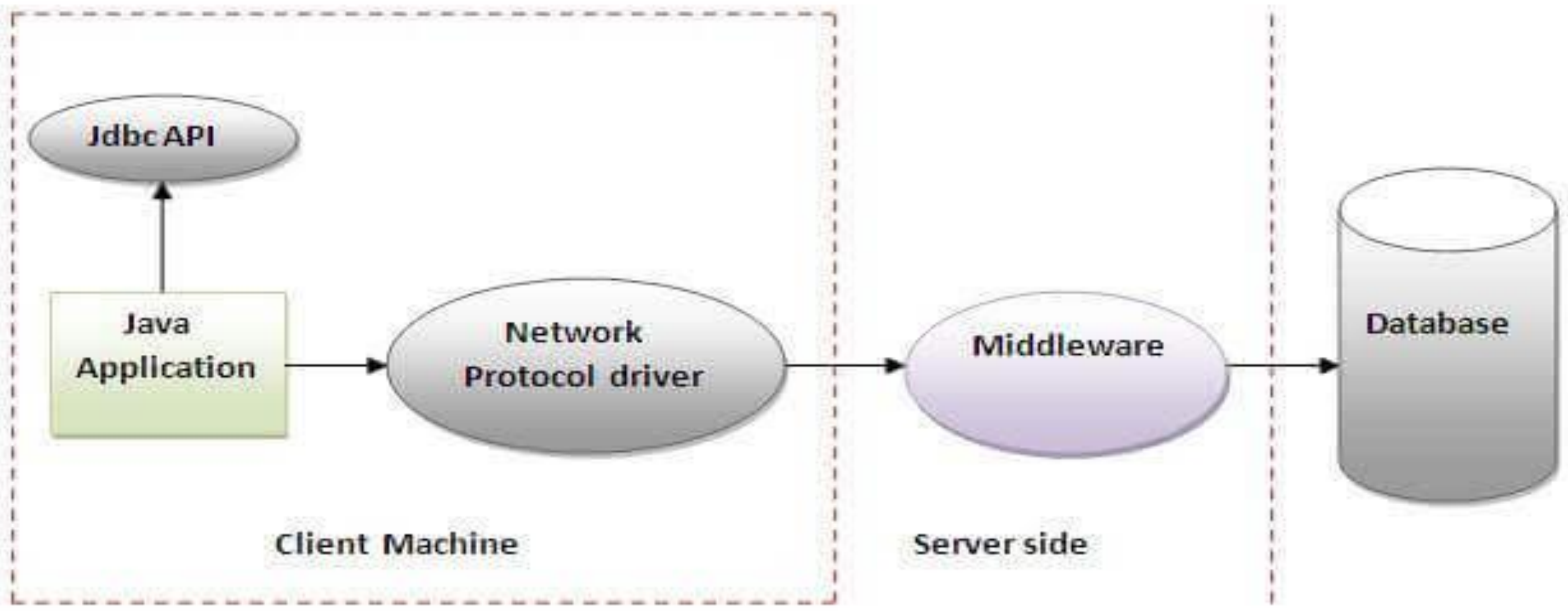


Figure- Network Protocol Driver

Advantages & Disadvantages

Advantages

1. Since the communication between client and the middleware server is database independent, there is no need for the database vendor library on the client. The client need not be changed for a new database.
2. Platform independent.
3. Popular for network applications
4. portable

Disadvantages

1. Network support is required on client machine.
2. Requires database-specific coding to be done in the middle tier.
3. Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

Type 4: Native protocol java Driver(Thin driver (fully java driver))

- The JDBC type 4 driver, also known as the Direct to Database **Pure Java Driver**, converts JDBC calls directly into a vendor-specific database protocol.
- Written completely in Java, type 4 drivers are thus platform independent. They install inside the Java Virtual Machine of the client. This provides better performance than the type 1 and type 2 drivers as it does not have the overhead of conversion of calls into ODBC or database API calls. Unlike the type 3 drivers, it does not need associated software to work.
- As the database protocol is vendor specific, the JDBC client requires separate drivers, usually vendor supplied, to connect to different types of databases.

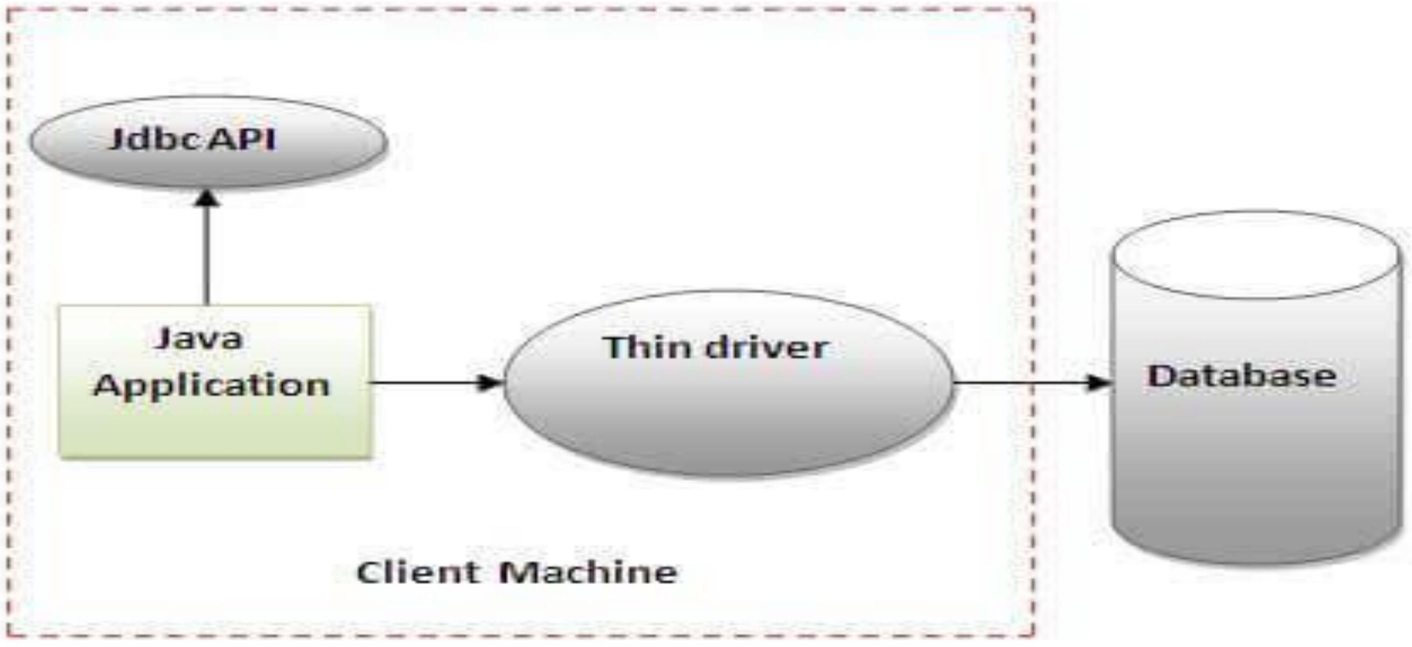


Figure- Thin Driver

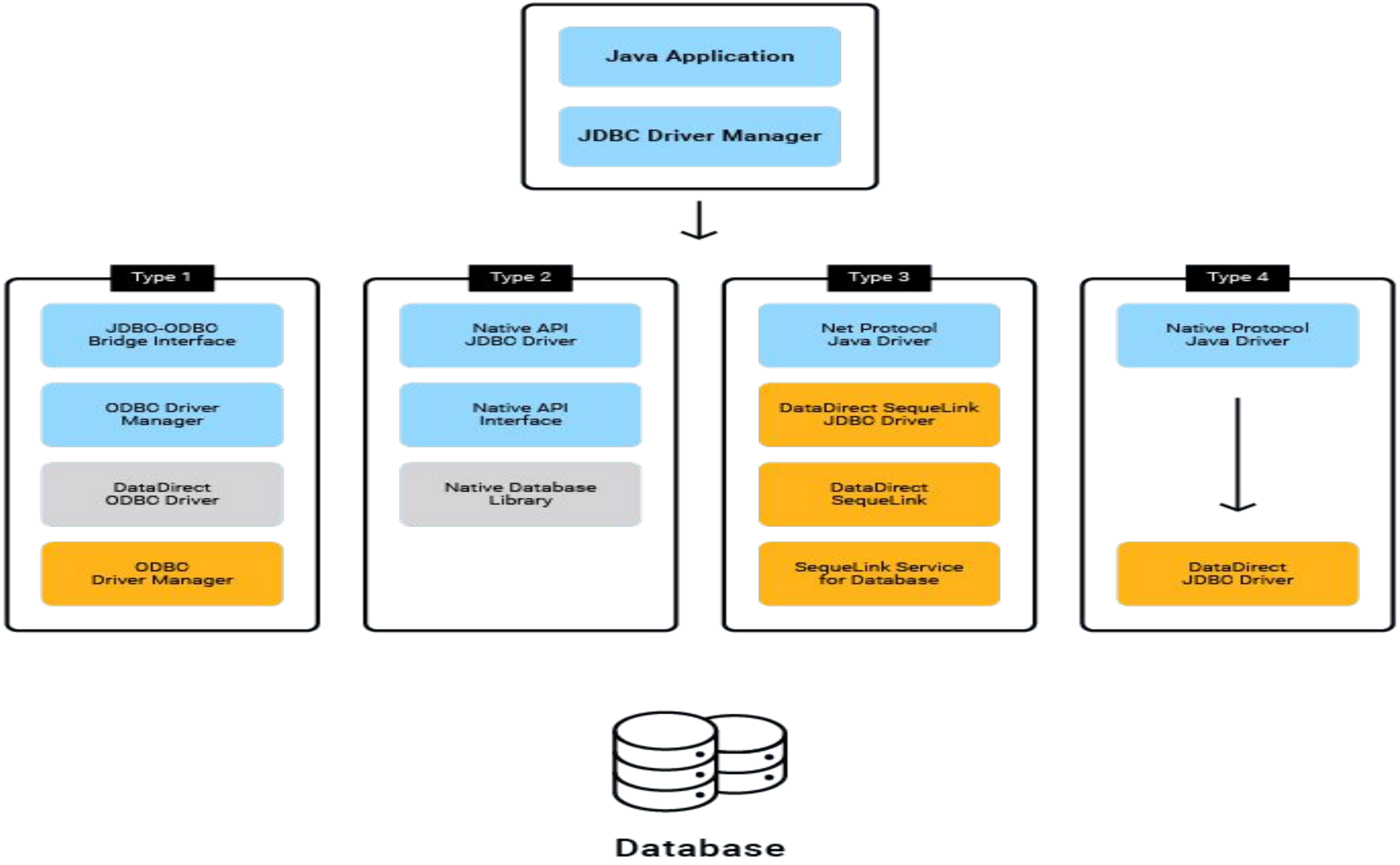
Advantages & Disadvantages

Advantages

1. Completely implemented in Java to achieve platform independence.
2. These drivers don't translate the requests into an intermediary format (such as ODBC).
3. The client application connects directly to the database server. No translation or middleware layers are used, improving performance.
4. The JVM can manage all aspects of the application-to-database connection; this can facilitate debugging.

Disadvantages

1. Drivers are database specific, as different database vendors use widely different (and usually proprietary) network protocols.


Java Application
JDBC Driver Manager
Type 1
JDBC-ODBC Bridge Interface
ODBC Driver Manager
DataDirect ODBC Driver
ODBC Driver Manager
Type 2
Native API JDBC Driver
Native API Interface
Native Database Library
Type 3
Net Protocol Java Driver
DataDirect SequeLink JDBC Driver
DataDirect SequeLink
SequeLink Service for Database
Type 4
Native Protocol Java Driver
DataDirect JDBC Driver
Database

Fundamental Steps in JDBC

The fundamental steps involved in the process of connecting to a database and executing a query consist of the following:

1. Import JDBC packages.
2. Load and register the JDBC driver.
3. Open a connection to the database.
4. Create a statement object to perform a query.
5. Execute the statement object and return a query resultset.
6. Process the resultset.
7. Close the resultset and statement objects.
8. Close the connection.

1. Import JDBC Packages
This is for making the JDBC API classes immediately available to the application program. The following import statement should be included in the program irrespective of the JDBC driver being used:
import java.sql.*;
2. Load and Register the JDBC Driver
You must register the driver in your program before you use it. Registering the driver is the process by which the driver's class file is loaded into the memory.
This is for establishing a communication between the JDBC program and the database.
Approach I: The forName() method of the java.lang.Class class can be used to dynamically load the driver's class file into memory, which automatically registers it.

- Class.forName("Driver_name");
- e.g
- Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");//for MS access
  Class.forName("com.mysql.jdbc.Driver");//mysql
- Class.forName("org.postgresql.Driver");//postgresql

- Approach II - DriverManager.registerDriver()
- The second approach you can use to register a driver, is to use the static DriverManager.registerDriver() method.
- You should use the registerDriver() method if you are using a non-JDK compliant JVM, such as the one provided by Microsoft.
- E.g
  Driver myDriver = new oracle.jdbc.driver.OracleDriver();
  DriverManager.registerDriver(myDriver);
- Or
- DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());

Connecting to a Database

- ✔ Once the required packages have been imported and the JDBC driver has been loaded and registered, a database connection must be established.
- ✔ This is done by using the getConnection()method of the DriverManager class. A call to this method creates an object instance of the java.sql.Connection class.

The getConnection() method is an overloaded method that takes

•Three parameters, one each for the URL, username, and password.

•Only one parameter for the database URL. In this case, the URL contains the username and password.

The following lines of code illustrate using the getConnection() method:

```
Connection conn = DriverManager.getConnection(URL, username, passwd);
Connection conn = DriverManager.getConnection(URL);
```

where URL, username, and passwd are of String data types.

```
Connection conn
=DriverManager.getConnection("jdbc:mysql://localhost/db_name");//mysql
Connection conn =
DriverManager.getConnection("jdbc:postgresql://localhost:5432/db_name");//postgresql
Connection conn = DriverManager.getConnection("jdbc:odbc:abc”);//access
```

RDBMS	JDBC driver name	URL format
MySQL	com.mysql.jdbc.Driver	**jdbc:mysql://**hostname/ databaseName
ORACLE	oracle.jdbc.driver.OracleDriver	**jdbc:oracle:thin:@**hostname:port Number:databaseName
DB2	COM.ibm.db2.jdbc.net.DB2Driver	**jdbc:db2:**hostname:port Number/databaseName
Sybase	com.sybase.jdbc.SybDriver	**jdbc:sybase:Tds:**hostname: port Number/databaseName
postgresql.	org.postgresql.Driver"	jdbc:postgresql://localhost:5432/testdb","postgres", "123"

Querying the database involves two steps: first, creating a statement object to perform a query, and second, executing the query and returning a resultset.

Creating a Statement Object

This is to instantiate objects that run the query against the database connected to.

There are three types of statements in JDBC

•**Statement**: This represents a general SQL statement without parameters.

•This is done by the createStatement() method of the conn Connection object created above.

•A call to this method creates an object instance of the Statement .

•Statement sql_stmt = conn.createStatement();

•**PreparedStatement**: This represents a precompiled SQL statement,with or without parameters. PrepareStatement are used for SQL commands that need to be executed repeatedly.

•The method prepareStatement(String sql)Creates a PrepareStatement object

•e.g. PreparedStatement ps=conn.prepareStatement("select * from student");

•**CallableStatement :** CallableStatement Objects are used to execute SQL stored procedures.

•The method preapareCall(String sql)creates a CallableStatementObject.

•e.g. CallableStatement cs=conn.prepareCall   ("{call show_students}");

Executing the Query and Returning a ResultSet

A SQL Statement can either be a query that returns result or an operation that manipulates the database.Following methods of the Statement interface are used to execute the query.

1) ResultSet executeQuery(String sql) : A call to this method takes as parameter a SQL SELECT statement and returns a JDBC ResultSet object.

•The following line of code illustrates this using the sql_stmt object created above:

•String query="SELECT empno, ename, sal, deptno FROM emp ORDER BY ename";

ResultSet rset = sql_stmt.executeQuery(“select * from student”);

2) int executeUpadate(String sql) : It is used for statements that do not return an output such as update, insert, create, delete etc.

It returns the no. of rows changed.

e.g. int result=sql_stmt.executeUpdate(“DROP TABLE test”);

3) **boolean execute(String sql):** executes query and returns true if statement Statement returns a result.

boolean result=sql_stmt.execute(“DROP TABLE test”);

Statement and ResultSet Objects

Statement and ResultSet objects open a corresponding cursor in the database for SELECT and other DML statements.
The above statement executes the SELECT statement specified in between the double quotes and stores the resulting rows in an instance of the ResultSet object named rset.

- A ResultSet object maintains a cursor pointing to its current row of data.
- Initially the cursor is positioned before the first row.
- To iterate the ResultSet you use its next() method.
- The next() method returns true if the ResultSet has a next record, and moves the ResultSet to point to the next record. If there were no more records, next() returns false
- E.g. while(rs.next())

- The ResultSet interface contains a getXXX() method to get database table data.
- Here XXX represents the datatype
- get method has two versions –
- One that takes in a column name.

  i.e getXXX(“Col_name”) ->getInt(“Eno”)
- One that takes in a column index.

  i.e getXXX(Col_number)->getInt(1)
- For example, if the column you are interested in viewing contains an int, you need to use one of the getInt() methods of ResultSet –

Closing the ResultSet and Statement

Once the ResultSet and Statement objects have been used, they must be closed explicitly.
This is done by calls to the close() method

```
rset.close();
sql_stmt.close();
```

If not closed explicitly, there are two disadvantages:

- Memory leaks can occur
- Maximum Open cursors can be exceeded

Closing the ResultSet and Statement objects frees the corresponding cursor in the database.

Closing the Connection

The last step is to close the database connection opened in the beginning after importing the packages and loading the JDBC drivers. This is done by a call to the close() method of the Connection class.

The following line of code does this:

```
conn.close();
```

Common JDBC Components

- The JDBC API provides the following interfaces and classes −
- **DriverManager:** This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub protocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.
- **Driver:** This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects.
- **Connection:** This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.
- **Statement:** You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.
- **ResultSet:** These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.
- **SQLException:** This class handles any errors that occur in a database application.

Connection Methods

1) public Statement createStatement(): creates a statement object that can be used to execute SQL queries.

2) public Statement createStatement(int resultSetType,int resultSetConcurrency): Creates a Statement object that will generate ResultSet objects with the given type and concurrency.

3) public void setAutoCommit(boolean status): is used to set the commit status.By default it is true.

4) public void commit(): saves the changes made since the previous commit/rollback permanent.

5) public void rollback(): Drops all changes made since the previous commit/rollback.

6) public void close(): closes the connection and Releases a JDBC resources immediately.

Statement methods

1) public ResultSet executeQuery(String sql): is used to execute SELECT query. It returns the object of ResultSet.

2) public int executeUpdate(String sql): is used to execute specified query, it may be create, drop, insert, update, delete etc.

3) public boolean execute(String sql): is used to execute queries that may return multiple results.

4) public int[] executeBatch(): is used to execute batch of commands.

```
import java.sql.*;// import pack
public class JDBCExample {
    public static void main(String[] args) {
                Class.forName("org.postgresql.Driver");// register driver


        try {
Connection conn = DriverManager.getConnection(“jdbc:postgresql://localhost:5432/pooja”,admin, 123);
            System.out.println("Connected to the PostgreSQL successfully.");
        Stāṭement st=conn.createStatement();// create statement
        st.executeUpdate(“create table student(rollno int primary key, name varchar(20), per float”);//execute query
   System.out.println("table created successfully”);
        } catch (SQLException e) {
            System.out.println("Connection failed: " + e.getMessage());
        }
st.close();
conn.close();
    }
}
```

```
import java.sql.*;// import pack
public class JDBCExample {
    public static void main(String[] args) {
              Class.forName("org.postgresql.Driver");// register driver
        try {
Connection conn =
DriverManager.getConnection(“jdbc:postgresql://localhost:5432/pooja”,admin, 123);
            System.out.println("Connected to the PostgreSQL successfully.");
       Stāṭement st=conn.createStatement();// create statement
       st.executeUpdate(“insert into Student values(1,’Sneha’,75)”);//execute query
   System.out.println("Value inserted successfully”);
        } catch (SQLException e) {
            System.out.println("Connection failed: " + e.getMessage());
        }
st.close();
conn.close();
    }
}
```

```
import java.sql.*;// import pack
public class JDBCExample {
  public static void main(String[] args) {
        Class.forName("org.postgresql.Driver");// register driver
    try {
Connection conn =
DriverManager.getConnection(“jdbc:postgresql://localhost:5432/pooja”,admin, 123);
         System.out.println("Connected to the PostgreSQL successfully.");
     Stātement st=conn.createStatement();// create statement
     st.executeUpdate(“update student set per=65 where rollno=1;”);//execute query
  System.out.println("updated successfully”);
    } catch (SQLException e) {
         System.out.println("Connection failed: " + e.getMessage());
    }
st.close();
conn.close();
   }
}
```

```
import java.sql.*;// import pack
public class JDBCExample {
    public static void main(String[] args) {
                Class.forName("org.postgresql.Driver");// register driver
        try {
Connection conn = DriverManager.getConnection(“jdbc:postgresql://localhost:5432/pooja”,admin, 123);
            System.out.println("Connected to the PostgreSQL successfully.");
        Stāṭement st=conn.createStatement();// create statement
        st.executeUpdate(“delete from studnt where rollno=1;”);//execute query
   System.out.println("table created successfully”);
        } catch (SQLException e) {
            System.out.println("Connection failed: " + e.getMessage());
        }
st.close();
conn.close();
    }
```

```
import java.sql.*;// import pack
public class JDBCExample {
    public static void main(String[] args) {
            Class.forName("org.postgresql.Driver");// register driver
        try {
Connection conn =
DriverManager.getConnection(“jdbc:postgresql://localhost:5432/pooja”
,admin, 123);
            System.out.println("Connected to the PostgreSQL successfully.");
      Stāṭement st=conn.createStatement();// create statement
      ResultSet rs=st.executeQuery(“select * from student”);
      while(rs.next())
      {
   System.out.println("Roll no:”+rs.getInt(1));
  System.out.println("Name:”+rs.getString(2));
  System.out.println("Per:”+rs.getFloat(3));
      }
       } catch (SQLException e) {
```

PreparedStatement interface

? The PreparedStatement interface is a subinterface of Statement. It is used to execute parameterized query.

? This statement gives you the flexibility of supplying arguments dynamically.

? Let's see the example of parameterized query:

```
String sql="insert into emp values(?,?,?)";
```

? All parameters in JDBC are represented by the **?** symbol, which is known as the parameter marker. You must supply values for every parameter before executing the SQL statement.

? The **setXXX()** methods bind values to the parameters, where **XXX** represents the Java data type of the value you wish to bind to the input parameter. If you forget to supply the values, you will receive an SQLException.

? Each parameter marker is referred by its ordinal position. The first marker represents position 1, the next position 2, and so forth. This method differs from that of Java array indices, which starts at 0.

PreparedStatement interface methods

Method	Description
public void setInt(int paramIndex, int value)	sets the integer value to the given parameter index.
public void setString(int paramIndex, String value)	sets the String value to the given parameter index.
public void setFloat(int paramIndex, float value)	sets the float value to the given parameter index.
public void setDouble(int paramIndex, double value)	sets the double value to the given parameter index.
public int executeUpdate()	executes the query. It is used for create, drop, insert, update, delete etc.
public ResultSet executeQuery()	executes the select query. It returns an instance of ResultSet.

The CallableStatement Objects

? Just as a Connection object creates the Statement and PreparedStatement objects, it also creates the CallableStatement object, which would be used to execute a call to a database stored procedure.

? Three types of parameters exist: IN, OUT, and INOUT.

Parameter	Description
IN	A parameter whose value is unknown when the SQL statement is created. You bind values to IN parameters with the setXXX() methods.
OUT	A parameter whose value is supplied by the SQL statement it returns. You retrieve values from theOUT parameters with the getXXX() methods.
INOUT	A parameter that provides both input and output values. You bind variables with the setXXX() methods and retrieve values with the getXXX() methods.

Resultset Types

- ? The object of ResultSet maintains a cursor pointing to a row of a table. Initially, cursor points to before the first row
- ? By default, ResultSet object can be moved forward only and it is not updatable.
- ? But we can make this object to move forward and backward direction by passing either TYPE_SCROLL_INSENSITIVE or TYPE_SCROLL_SENSITIVE in createStatement(int,int) method as well as we can make this object as updatable by:
- ? Statement stmt = con.createStatement(int scroll_type, int concurrency_type);

Scroll type

Type	Description
ResultSet.TYPE_FORWARD_ONLY	The cursor can only move forward in the result set.
ResultSet.TYPE_SCROLL_INSENSITIVE	The cursor can scroll forward and backward, and the result set is not sensitive to changes made by others to the database that occur after the result set was created.
ResultSet.TYPE_SCROLL_SENSITIVE.	The cursor can scroll forward and backward, and the result set is sensitive to changes made by others to the database that occur after the result set was created.

Concurrency type

Concurrency	Description
ResultSet.CONCUR_READ_ONLY	Creates a read-only result set. This is the default
ResultSet.CONCUR_UPDATABLE	Creates an updateable result set.

ResultSet interface methods

Method	Purpose
1) public boolean next():	is used to move the cursor to the one row next from the current position.
2) public boolean previous():	is used to move the cursor to the one row previous from the current position.
3) public boolean first():	is used to move the cursor to the first row in result set object.
4) public boolean last():	is used to move the cursor to the last row in result set object.
5) public boolean absolute(int row):	is used to move the cursor to the specified row number in the ResultSet object.

Method	Purpose
6) public boolean relative(int row):	is used to move the cursor to the relative row number in the ResultSet object, it may be positive or negative.
7) public int getInt(int columnIndex):	is used to return the data of specified column index of the current row as int.
8) public int getInt(String columnName):	is used to return the data of specified column name of the current row as int.
9) public String getString(int columnIndex):	is used to return the data of specified column index of the current row as String.
10) public String getString(String columnName):	is used to return the data of specified column name of the current row as String.

? To update your changes to the row in the database, you need to invoke one of the following methods.

S.N.	Methods & Description
1	**public void updateRow()**Updates the current row by updating the corresponding row in the database.
2	**public void deleteRow()**Deletes the current row from the database
3	**public void refreshRow()**Refreshes the data in the result set to reflect any recent changes in the database.
4	**public void cancelRowUpdates()**Cancels any updates made on the current row.
5	**public void insertRow()**Inserts a row into the database. This method can only be invoked when the cursor is pointing to the insert row.
6	The **getRow()** method of the **ResultSet** interface retrieves the current row number/position of the ResultSet pointer.

insertRow

? To insert a record into a ResultSet using the insertRow() method −

1. Create the ResultSet object
2. The moveToInsertRow() method of the ResultSet interface navigates the cursor to the position where you need to insert the next record. Therefore, move the cursor to the appropriate position to insert a row using this method.
3. The updateXXX() methods of the ResultSet interface allows you to insert/update values in to the ResultSet object.

? E.g

```
rs.moveToInsertRow();
rs.updateInt(1, 220);
rs.updateString(2, "sdadsads");
```

updateRow()

? The **updateRow()** method of the **ResultSet** interface updates the contents of the current row to the database. For example if we have updated the values of a particular record using the updateXXX() methods of the ResultSet interface (updateNClob(), updateNCharacterStream(), updateString(), updateInt(), updateNString(), updateBinaryStream())

? You need to invoke this method to reflect the changes you have made to the row in the database.

? E.g

? rs.updateString(2, "Name");

? rs.updateRow();

deleteRow()

- ? The deleteRow() method of the ResultSet interface deletes the current row from the ResultSet object and from the table.
- ? E.g

```
rs.deleteRow();
```

MetaData

- ? What is Meta Data?
- ? Metadata is data about the data or documentation about the information which is required by the users.
- ? Types of MetaData in Java

1. DatabaseMetaData: DatabaseMetaData interface provides methods to get meta data of a database such as database product name, database product version, driver name, name of total number of tables, name of total number of views etc.
2. ResultSetMetaData: ResultSetMetaData interface is useful because it provides methods to get metadata from the ResultSet object.
3. ParameterMetaData: It can be used to get information about the types and properties for each parameter marker in a PreparedStatement and CallableStatement object

Java DatabaseMetaData interface

? DatabaseMetaData interface provides methods to get meta data of a database such as database product name, database product version, driver name, name of total number of tables, name of total number of views etc.

? DatabaseMetaData interface methods

? public String getDriverName()throws SQLException: it returns the name of the JDBC driver.

? public String getDriverVersion()throws SQLException: it returns the version number of the JDBC driver.

? public String getUserName()throws SQLException: it returns the username of the database.

? public String getDatabaseProductName()throws SQLException: it returns the product name of the database.

- public String getDatabaseProductVersion()throws SQLException**:** it returns the product version of the database.
- public ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types)throws SQLException: it returns the description of the tables of the specified catalog. The table type can be TABLE, VIEW, ALIAS, SYSTEM TABLE, SYNONYM etc.
- ResultSet getPrimaryKeys(String catalog, String schema, String table);
- ResultSet getImportedKeys(String catalog, String schema, String table);
- ResultSet getExportedKeys(String catalog, String schema, String table);
- ResultSet getProcedures(String catalog, String schemaPattern,String procedureNamePattern);
- ResultSet getProcedureColumns(String catalog, String schemaPattern, String procedureNamePattern, String columnNamePattern);
- How to get the object of DatabaseMetaData:
- The getMetaData() method of Connection interface returns the object of DatabaseMetaData.

ResultSetMetaData

? The **ResultSetMetaData** provides information about the obtained ResultSet object like, the number of columns, names of the columns, datatypes of the columns, name of the table etc…

? Following are some methods of **ResultSetMetaData** class.

Method	Description
getColumnCount()	Retrieves the number of columns in the current ResultSet object.
getColumnLabel()	Retrieves the suggested name of the column for use.
getColumnName()	Retrieves the name of the column.
getTableName()	Retrieves the name of the table.