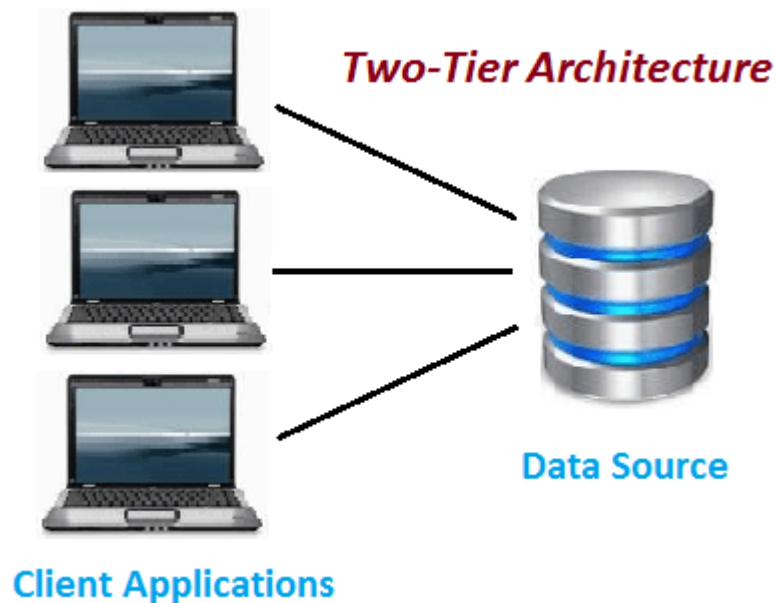


Two-Tier Database Design.

- The two-tier is based on **Client-Server** architecture.
- The direct communication takes place between client and server.
- There is no mediator between client and server.
- Because of **tight coupling** a 2 tiered application will run faster.



- Example, Railway reservation application software.
- In this application, both **Database** and **Server** are incorporated with each other, so this technology is called as “**Client-Server Technology**”.
- The Two-tier architecture is divided into two parts:
 - 1) **Client Application** (Client Tier / presentation layer)
 - 2) **Database** (Data Tier / data layer)
- On client application side the code is written for saving the data in the SQL server database. Client sends the request to server and it process the request & send back with data.

Advantages:

- High **portability**.
- Systems are accessible by **multiple users** from any part of the world
- Easy to **maintain** and **modification** is bit easy as compare to 3 - tier.
- Communication is **faster**.
- Due to their **less complexity**, easy to build and thereby less **expensive**.

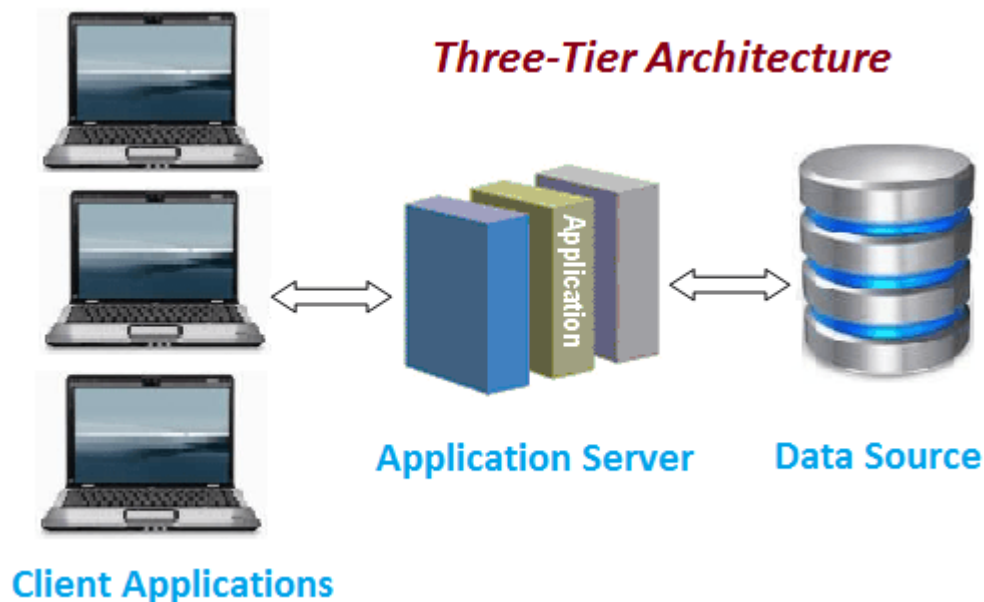
Disadvantages:

- Performance is reduced when **clients increase**. But generally faster with **less number of users** due to **tight coupling** between client and server.

- More suitable to **departmental applications** with small scale groupware and simple for Web-based.

Three-Tier Database Design.

- Three-tier architecture typically comprises **Presentation tier**, **Business tier** or **data access tier**, and **Data tier**.
- Three layers in the three tier architecture are as follows:
 - 1) **Client layer** (Presentation Layer)
 - 2) **Business layer** (Application Layer / Middle Tier)
 - 3) **Data layer**



1) Client Layer :

- It is also called as **Presentation Layer** which contains **UI** part of our application.
- This layer is used for the **design** purpose where data is presented to the user or input is taken from the user.
- For this, in web communication **HTML** is used.

2) Business Layer :

- In this layer all **business logic** written like validation of data, calculations, data insertion etc.
- It acts as an **interface** between **Client Layer and Data Layer**.
- It is also called the **mediator**, helps to make **communication faster** between client and data layer.

3) Data Layer :

- In this layer **actual database** is comes in the picture.
- It contains **methods to connect with database** and to perform insert, update, delete, get data from database based on our input data.

Advantages:

- **Improved Security** – Client is not direct access to database.
- **Scalability** – Each layer is run on different systems.
- **Performance** is higher than 2-tier.
- **High degree of flexibility** as an additional tier for integration logic.
- It is **fast in communication** than 2-tier.
- Improve **Data Integrity**.
- Easy to **maintain** and **modification** is bit easy, won't affect other modules.
- Proved the possibility of **integration** of servers where one server can communicate with other servers.

Disadvantages:

- Increase **Complexity/Effort**

JDBC Drivers

- JDBC Driver is a **software component** that enables java application to interact with the database.
- There are **four types of JDBC drivers** :

- 1) **Type - 1** : JDBC-ODBC Bridge driver (Bridge)
- 2) **Type - 2** : Native-API/partly Java driver (Native)
- 3) **Type - 3** : All Java/Net-protocol driver (Middleware)
- 4) **Type - 4** : All Java/Native-protocol driver (Pure)

1) JDBC-ODBC Bridge driver

- In Type - 1 driver, a JDBC bridge is used to access ODBC drivers installed on each client machine.
- Using ODBC, requires configuring on your system a Data Source Name (DSN) that represents the target database.
- The Type 1 driver translates all **JDBC calls** into **ODBC calls** and sends them to the **ODBC driver**.
- **ODBC** is a generic **API**.
- The JDBC-ODBC Bridge driver is recommended only for experimental use or when no other alternative is available.

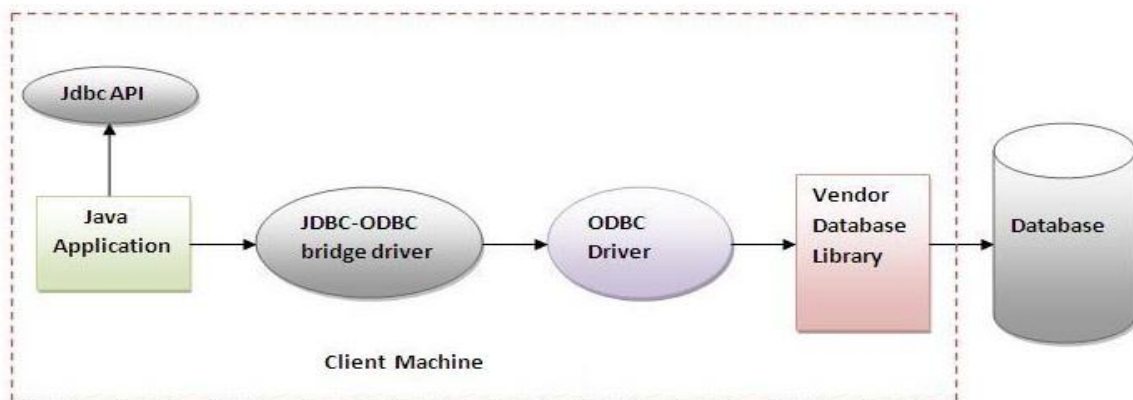


Figure- JDBC-ODBC Bridge Driver

Advantages:

- Easy to use.
- Can be easily **connected to any database**.

Disadvantages:

- The Bridge driver is not written fully in Java, Type 1 drivers are **not portable**.
- **Performance** degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver needs to be **installed** on the **client machine**.
- **Not good** for the **Web**.
- It is the **slowest** of all driver types.

2) Native-API/partly Java driver

- Type 2 drivers convert **JDBC calls into database-specific calls**.
- This driver is specific to a particular database.
- The vendor-specific driver must be installed on each client machine.
- If we change the Database, we have to change the native API because it is database specific.
- It is not written entirely in java.

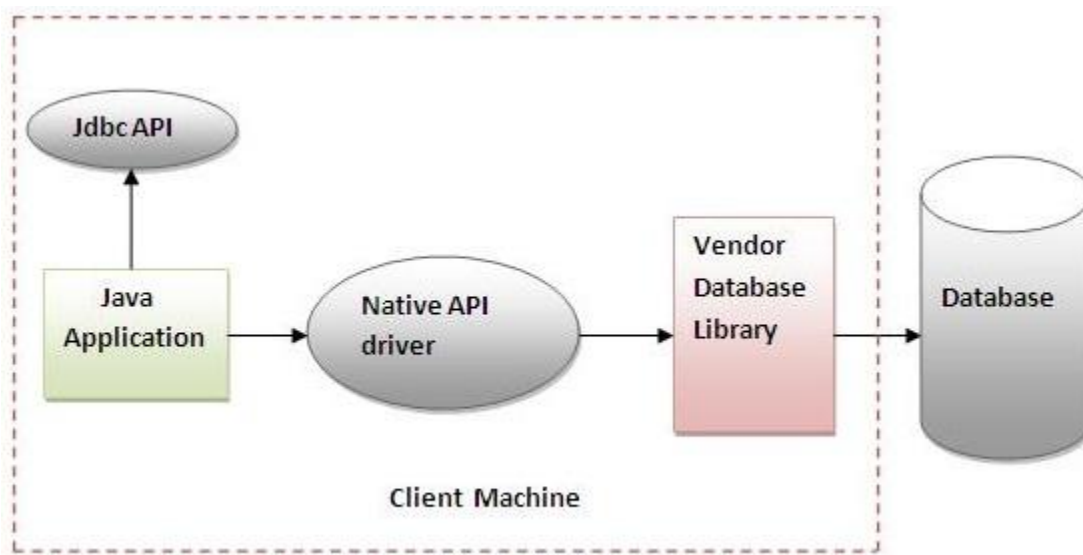


Figure- Native API Driver

Advantages:

- **Performance increases** with a Type 2 driver as compare to Type 1 driver, because it eliminates ODBC's overhead.

Disadvantages:

- The **Native driver** needs to be installed on the each client machine.
- The **Vendor client library** needs to be installed on client machine.
- If we change the Database we have **to change the native API** as it is specific to a database.
- It is **platform dependent**.

3) Pure Java/Network-protocol/Middleware driver

- Type 3 driver is used **three-tier** approach to access databases.
- It uses **middleware** (application server) that converts **JDBC calls** directly or indirectly into the **vendor-specific** database protocol.
- It is fully written in **java**.

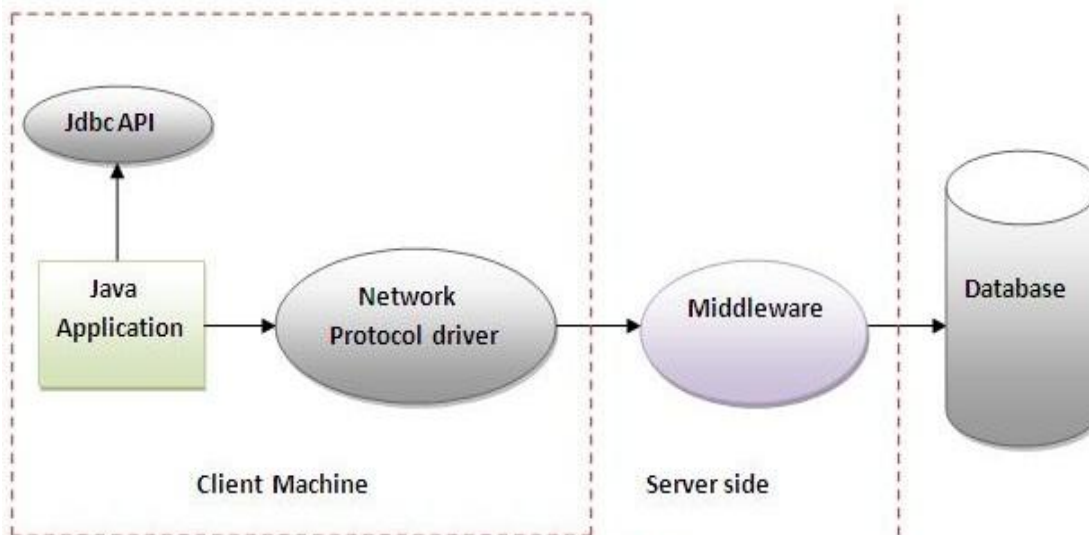


Figure- Network Protocol Driver

Advantages:

- This driver is **server-based**, so there is no need for any **vendor database library** to be present on client machines.
- It is **portable** because it is fully written in java.
- It is very **flexible** allows access to multiple databases using one driver.
- They are **the most efficient** amongst all driver types.
- The middleware server can provide typical middleware services like **caching, load balancing, logging** and **auditing**.

Disadvantages:

- **Network support** is required on client machine.
- **Maintenance** of Network Protocol driver becomes **costly** because it requires **database-specific coding** to be done in the middle tier.
- It requires another server application to **install** and **maintain**.

4) Database-protocol/Pure Java driver

- Type 4 driver is an all Java driver which connects **directly to the database**. It is also known as **Thin Driver**.
- It is a database driver implementation that **converts JDBC calls** directly into a **vendor-specific database protocol**.
- The database protocol is vendor specific, the JDBC client requires separate drivers, usually vendor supplied, to connect to different types of databases.

- It is fully written in **Java** language.

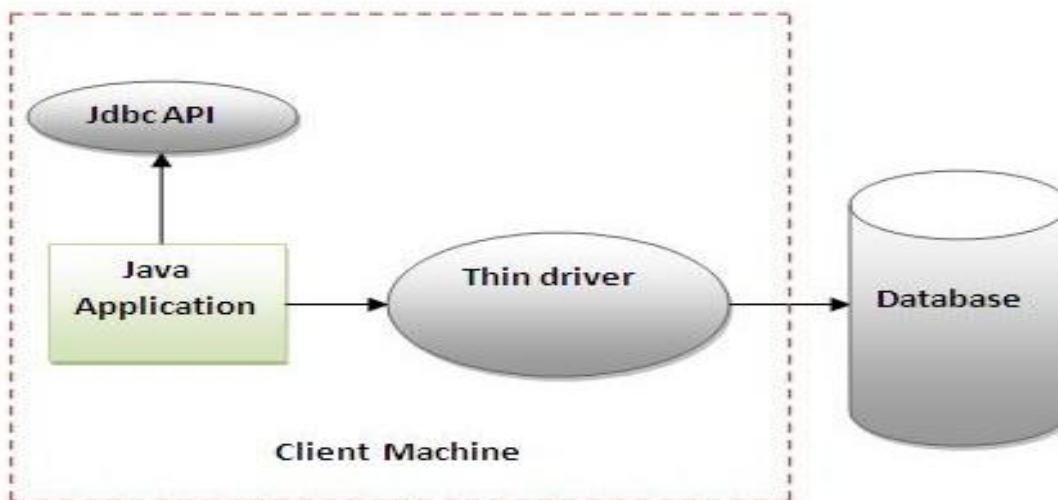


Figure- Thin Driver

Advantages:

- It is **platform dependent**.
- The client application connects directly to the database server. No **translation** or **middleware** layers are used, performance is quite good.
- **No software** is required at client side or server side.

Disadvantages:

- Drivers depend on the Database.

JDBC-ODBC Bridge

- The JDBC-ODBC Bridge is a JDBC driver which implements **JDBC operations by translating them into ODBC operations**.
- To ODBC it appears as a normal **application program**.
- The bridge implements JDBC for any database for which an **ODBC driver is available**.
- The bridge is implemented as the **sun.jdbc.odbc** Java package and contains a native library used to access ODBC.
- The bridge is a joint development of **Intervolve and Java Soft**.
- The bridge is implemented in **Java** and uses **Java native methods** to call ODBC.
- The bridge is installed **automatically** with the JDK as package **sun.jdbc.odbc**.
- **No special configuration** is required for the bridge.
- The bridge is used by opening a **JDBC connection** using a URL with the **odbc** subprotocol.
`Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");`
- Before a connection can be established, the bridge driver class, **sun.jdbc.odbc.JdbcOdbcDriver** must be **explicitly loaded** using the **Java class loader**.
- When loaded, the ODBC driver **creates an instance of itself** and **registers** this with the **JDBC driver manager**.

- JDBC used with a **Pure Java JDBC driver** works well with **applets**. The bridge driver does **not work** well with **applets**.
- The bridge driver uses the **odbc subprotocol**. URLs for this subprotocol are of the form:
jdbc:odbc:<data-source-name>[<attribute-name>=<attribute-value>]*

Example:

```
jdbc:odbc:sybase
jdbc:odbc:mydb;UID=me;PWD=secret
```

JDBC API

- JDBC API provides **the application to JDBC Manager connection**.
- The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to **heterogeneous databases**.
- Some of the important **classes** and **interface** defined in JDBC API are as follows.

Class / Interface	Description
Driver Manager	The DriverManager class loads and configures a database driver on the client side.
Connection	<p>The Connection class performs connection and authentication to a database server.</p> <p>Syntax:</p> <pre>Connection con = DriverManager.getConnection (url, "username", "password");</pre>
Statement	<p>Useful when you are using static SQL statement at runtime and return the results by using ResultSet object. The Statement interface cannot accept parameter. The object is created using the createStatement() method of the Connection interface.</p> <p>Syntax:</p> <pre>Connection con = DriverManager.getConnection (url, "username", "password"); Statement stmt = con.createStatement();</pre>
PreparedStatement	<p>The PreparedStatement interface accepts input parameter at runtime. It is subclass of the Statement interface. It is precompiled query which can be executed multiple times. The object is created using the prepareStatement() method of Connection interface.</p> <p>Syntax:</p> <pre>Connection con = DriverManager.getConnection (url, "username", "password"); String query = "insert into emp values(?, ?)"; PreparedStatement ps = con.prepareStatement(query);</pre>

	<pre>ps.setInt(1,5); ps.setString(2, "New Employee"); int n = ps.executeUpdate();</pre>
CallableStatement	<p>The CallableStatement interface can also accept input parameter at runtime. It is used to call the stored procedures and functions. The object is created using the prepareCall() method of connection interface.</p> <p>Syntax:</p> <pre>Connection cn = DriverManager.getConnection("jdbc:odbc:exm"); CallableStatement cs = cn.prepareCall("{call proc1()}"); ResultSet rs= cs.executeQuery();</pre>
ResultSet	<p>The object of ResultSet maintains a cursor pointing to a particular row of data. Initially, cursor points to before the first row. You can use the next() method of ResultSet to move from row to row.</p>
DatabaseMetaData	<p>DatabaseMetaData interface provides methods to get meta data of a database such as database product name, database product version, driver name, driver version etc.</p>
ResultSetMetaData	<p>ResultSetMetaData interface provides methods to get meta data of a Resultset such as table name, column name, column type, total number of column etc.</p>

JDBC Advantages and Disadvantages

Advantages:

- Can read any database if proper drivers are installed.
- No content conversion required.
- Query and Stored procedure supported.
- Can be used for both Synchronous and Asynchronous processing.
- Zero Configurations for Network Computers.
- Database Connection Identified by URL.

Disadvantages:

- Correct drivers need to be deployed for each type of database.
- Cannot update or insert multiple tables with sequence.

Develop java program using JDBC.

```
import java.sql.*;

public class DemoDatabase
{
    // JDBC driver name and database URL
    static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
    static final String DB_URL = "jdbc:mysql://localhost:3306/Student";

    // Database credentials
    static final String USER = "root";
    static final String PASS = "root";

    public static void main(String[] args)
    {
        Connection conn = null;
        Statement stmt = null;
        try
        {
            //STEP 2: Register JDBC driver
            Class.forName("com.mysql.jdbc.Driver").newInstance();

            //STEP 3: Open a connection
            System.out.println("Connecting to database...");
            conn = DriverManager.getConnection(DB_URL,USER,PASS);

            //STEP 4: Execute a query
            System.out.println("Creating statement...");
            stmt = conn.createStatement();
            String sqlQuery;
            sqlQuery = "SELECT Id, Name, Branch FROM demo";
            ResultSet rs = stmt.executeQuery(sqlQuery);

            //STEP 5: Extract data from result set
            while(rs.next())
            {
                //Retrieve by column name
                int id = rs.getInt("Id");
                String Name = rs.getString("Name");
                String Branch = rs.getString("Branch");
                //Display values
            }
        }
    }
}
```

```
        System.out.print("ID: " + id + "\n");
        System.out.print("Name: " + Name + "\n");
        System.out.print("Branch: " + Branch + "\n");
    }
    //STEP 6: Clean-up environment
    rs.close();
    stmt.close();
    conn.close();
}
catch(SQLException se)
{
    se.printStackTrace();
}
catch(Exception e)
{
    e.printStackTrace();
}
finally
{
    try
    {
        if(stmt!=null)
            stmt.close();
    }
    catch(SQLException se2)
    {
    }
    try
    {
        if(conn!=null)
            conn.close();
    }
    catch(SQLException se)
    {
        se.printStackTrace();
    }
}
System.out.println("It's over !!!");
}
```