JDBC

Short for Java Database Connectivity, a Java API that enables Java programs to execute SQL statements. This allows Java programs to interact with any SQL-compliant database.

The Java Database Connectivity (JDBC) API is the industry standard for database-independent connectivity between the Java programming language and a wide range of databases  SQL databases and other tabular data sources, such as spreadsheets or flat files. The JDBC API provides a call-level API for SQL-based database access.

Since nearly all relational database management systems (DBMSs) support SQL, and because Java itself runs on most platforms, JDBC makes it possible to write a single database application that can run on different platforms and interact with different DBMSs.

JDBC was developed by JavaSoft, a subsidiary of Sun Microsystems.

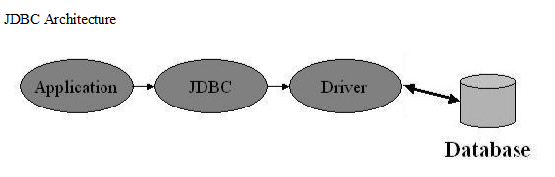
**JDBC Technology Drivers**

To use the JDBC API with a particular database management system, you need a JDBC technology-based driver to mediate between JDBC technology and the database. Depending on various factors, a driver might be written purely in the Java programming language or in a mixture of the Java programming language and JavaTM Native Interface (JNI) native methods.

The latest JDK includes a JDBC-ODBC Bridge driver that makes most Open Database Connectivity (ODBC) drivers available to programmers using the JDBC API

Note that the bridge driver included in the JavaTM 2 Platform Standard Edition 5.0 (J2SETM) is appropriate only for experimental use or when no other driver is available.

JDBC Architecture



Java application calls the JDBC library. JDBC loads a driver which talks to the

database

In general, to process any SQL statement with JDBC, you follow these steps:

①Establishing a connection.

②Create a statement.

③Execute the query.

④Process the ResultSet object.

⑤Close the connection.

<http://www.javatpoint.com/jdbc-driver>

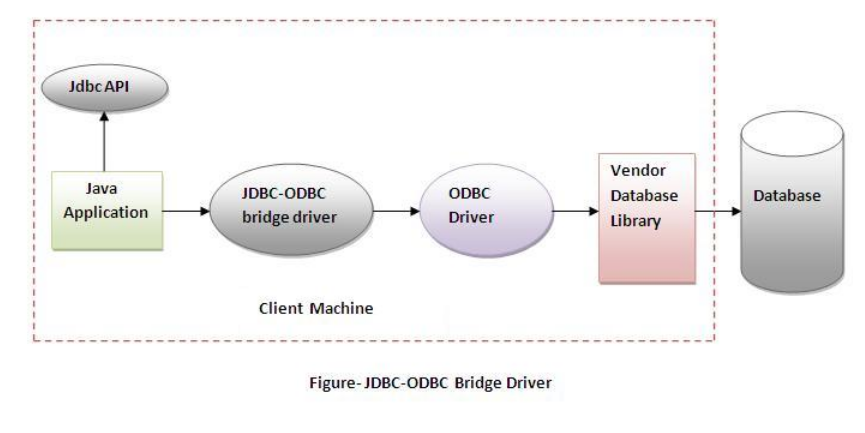
**Types of JDBC drivers :**

  There are four types of JDBC drivers known as:

* JDBC-ODBC bridge driver, also called *Type 1*.
* Native-API, partially Java driver, also called *Type 2.*
* JDBC-Network Protocol driver, pure/full Java driver, also called *Type 3.*
* Native-protocol, pure Java driver, also called *Type 4.*

**JDBC-ODBC bridge driver, also called *Type 1*:**

The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. It converts JDBC method calls into ODBC function calls .This is now discouraged because we now have thin driver (type4) which is much better.



Advantages:

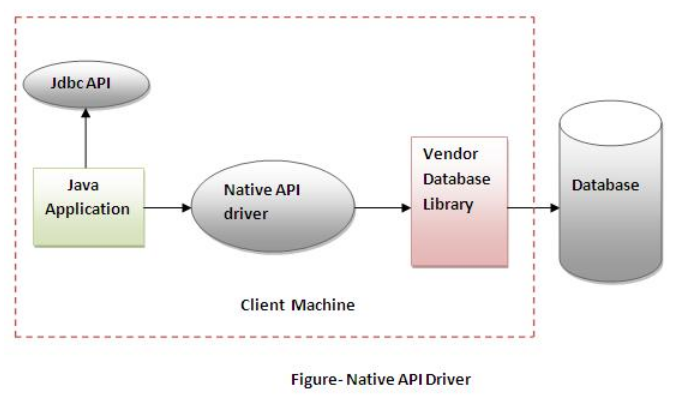
* Easy to use
* Can easily be connected to any database

Disadvantages:

* Performance is degraded because JDBC method calls is converted into ODBC function calls
* ODBC Driver need to be installed n client machine

**Native-API, partially Java driver, also called *Type 2.***

The Native-API uses the client side libraries of the database.The driver converts JDBC method calls of the database API.It is not written entirely in java.



Advantages:

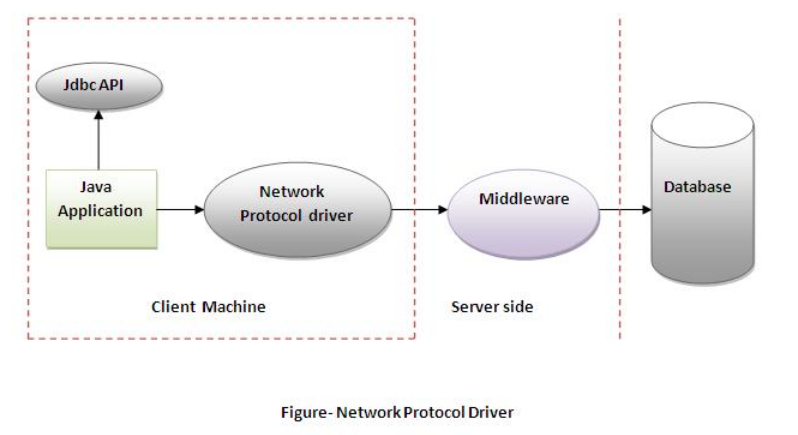
* Performance upgraded than JDBC-ODBC bridge driver
* Can easily be connected to any database

Disadvantages:

* The Native driver needs to be installed on each client machine.
* The Vendor client library needs to be installed on client machine.

**JDBC-Network Protocol driver, pure/full Java driver, also called *Type 3.***

It uses middleware(application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in JAVA.



Advantages:

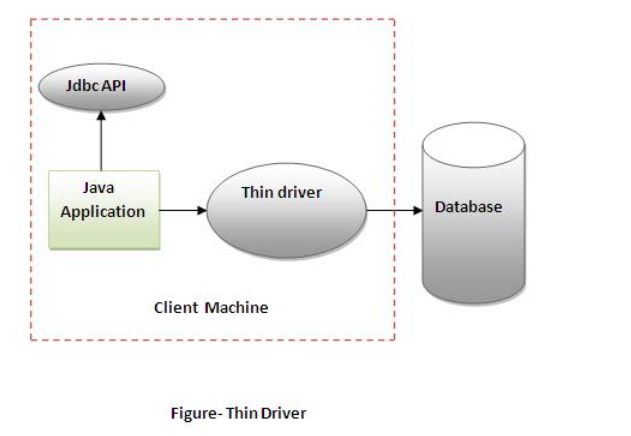
* No client side library is required because of application server –that can perform many tasks like ,auditing ,load balancing ,logging etc.

Disadvantages:

* Network support is required on client machine.
* Requires database specific coding to be done in the middle tier
* Maintenance of network protocol driver becomes costly because it requires database –specific coding to be done in middle tier.

***Native-protocol, pure Java driver, also called Type 4.(Thin Driver)***

The thin driver converts JDBC calls directly into the vendor specific database protocol. That is why it’s known as thin driver. It is fully written in java language.



Advantages:

* Better performance than all drivers
* No software required at client or server side.

Disadvantages:

* Driver depends on database.

**The JDBC API supports both two-tier and three-tier processing models for database access.**

**Figure 1: Two-tier Architecture for Data Access.**



In the two-tier model, a Java applet or application talks directly to the data source. This requires a JDBC driver that can communicate with the particular data source being accessed. A user's commands are delivered to the database or other data source, and the results of those statements are sent back to the user. The data source may be located on another machine to which the user is connected via a network. This is referred to as a client/server configuration, with the user's machine as the client, and the machine housing the data source as the server. The network can be an intranet, which, for example, connects employees within a corporation, or it can be the Internet.

**Figure 2: Three-tier Architecture for Data Access.**



In the three-tier model, commands are sent to a "middle tier" of services, which then sends the commands to the data source. The data source processes the commands and sends the results back to the middle tier, which then sends them to the user. MIS directors find the three-tier model very attractive because the middle tier makes it possible to maintain control over access and the kinds of updates that can be made to corporate data. Another advantage is that it simplifies the deployment of applications. Finally, in many cases, the three-tier architecture can provide performance advantages.

**JDBC Driver Manager** —  The JDBC DriverManager class defines objects which can connect Java applications to a JDBC driver. DriverManager has traditionally been the backbone of the JDBC architecture. It is quite small and simple. On running an application, it is the DriverManager's responsibility to load all the drivers found in the system property jdbc. drivers.

A **ResultSet** is a Java object that contains the results of executing an SQL query. In other words, it contains the rows that satisfy the conditions of the query. The data stored in a ResultSet object is retrieved through a set of get methods that allows access to the various columns of the current row. The ResultSet.next method is used to move to the next row of the ResultSet, making it the current row.

A **Statement** object is used to send SQL statements to a database. There are actually three kinds of Statement objects, all of which act as containers for executing SQL statements on a given connection: Statement, PreparedStatement, which inherits from Statement, and CallableStatement, which inherits from PreparedStatement. They are specialized for sending particular types of SQL statements; a Statement object is used to execute a simple SQL statement with no parameters, a PreparedStatement object is used to execute a precompiled SQL statement with or without IN parameters, and a CallableStatement object is used to execute a call to a database stored procedure.

The **PreparedStatement** interface inherits from Statement and differs from it in two ways:

1. Instances of PreparedStatement contain an SQL statement that has already been compiled. This is what makes a statement "prepared."
2. The SQL statement contained in a PreparedStatement object may have one or more IN parameters. An IN parameter is a parameter whose value is not specified when the SQL statement is created. Instead, the statement has a question mark ("?") as a placeholder for each IN parameter. The "?" is also known as a parameter marker. An application must set a value for each question mark in a prepared statement before executing the prepared statement.

A **CallableStatement** object provides a way to call stored procedures in a standard way for all RDBMSs. A stored procedure is stored in a database; the call to the stored procedure is what a CallableStatement object contains. This call is written in an escape syntax that may take one of two forms: one form with a result parameter, and the other without one. A result parameter, a kind of OUT parameter, is the return value for the stored procedure. Both forms may have a variable number of parameters used for input (IN parameters), output (OUT parameters), or both (INOUT parameters). A question mark serves as a placeholder for a parameter.

Batch Processing allows you to group related SQL statements into a batch and submit them with one call to the database.

When you send several SQL statements to the database at once, you reduce the amount of communication overhead, thereby improving performance.

* JDBC drivers are not required to support this feature. You should use the *DatabaseMetaData.supportsBatchUpdates()* method to determine if the target database supports batch update processing. The method returns true if your JDBC driver supports this feature.
* The **addBatch()** method of *Statement, PreparedStatement,* and *CallableStatement* is used to add individual statements to the batch. The **executeBatch()** is used to start the execution of all the statements grouped together.
* The **executeBatch()** returns an array of integers, and each element of the array represents the update count for the respective update statement.
* Just as you can add statements to a batch for processing, you can remove them with the **clearBatch()** method. This method removes all the statements you added with the addBatch() method. However, you cannot selectively choose which statement to remove.