Java-based approaches for accessing databases on the Internet and a JDBC-ODBC implementation

by Huiwei Guan, Horace H. S. Ip and Yanchun Zhang

An important area of Java development is to create Java-based software or tools with the capability of accessing databases over the Internet. In this article JDBC and its various properties are explored. Several Java-based technical approaches for accessing databases on the Internet are proposed and investigated in detail. A design and implementation of the Java software for accessing databases on the Internet is proposed and described, where the design scheme of an interactive applet by using Java JDBC programming is proposed and investigated, the main JDBC classes used in the Java program and the relationship between them are discussed, the implementation of the software is provided, and some instances for running the Java program are given. This Java software provides an interactive and user-friendly interface on the browser which supports Java JDK 1.1 or later and the software has the capability for the connection, SQL access, retrieval and display of a database on the Internet.

During the past few years, the computing industry has seen an incredible explosion in the growth of the Internet and the World Wide Web (WWW). In this relatively short period of time, the WWW has evolved from delivering simple static Web pages to the availability of custom dynamic Web pages created through various extensions to the Web servers. Although these techniques have enabled the delivery of data from corporate data stores, they have been limited both in their performance and their scalability.

Java offers the promise of reduced maintenance and support of application software and hardware. Java applets are mainly only installed and configured on a central server. These applets would then be downloaded on demand into browsers on client machines. With this methodology, no software is pre-installed on client machines, there are minimum software maintenance and distribution costs, and the applet will execute on any vendor's hardware supporting the browser.[1]

It is well known that many vendors have adopted the Java programming language more quickly than any other programming language in history. Industry, government and academia recognise the Java platform as a key technology for supporting platform-independent access to a wide range of information resources.[2,3]

The next step in this evolution is the use of Java to create applets to enable the missions in critical business applications and transaction processing over the Internet. This stage will require new high-performance and robust methods to access the corporate data stores.[4,5] Javasoft is leading the way in the development environment of libraries to extend the functionality and usability of Java as a serious platform for creating database applications. One of these libraries, as an application programming interface (API), is the Java Database Connectivity API (JDBC). Its primary purpose is to intimately tie connectivity to databases with the Java language.

In this article, the background of JDBC and its properties are briefly explored. Several technical

approaches for accessing databases on the Internet are then proposed and investigated. A design and implementation of Java software for accessing databases on the Internet is then proposed and described, where the programming technique of JDBC, JDBC-ODBC bridge, and ODBC driver are used, the main JDBC classes used in the Java software and the relationship between them are discussed, the design scheme and the implementation are proposed, and some instances for running the Java software are given.

JDBC background and properties

JDBC stands for Java Database Connectivity and it is a Java API for executing SQL statements. It consists of a set of classes and interfaces written in the Java programming language. JDBC provides a standard API for tool/database developers and makes it possible to write database applications by using a pure Java API.

The JDBC project was started in January 1996, and the specification was frozen in June 1996.[6] Javasoft sought the input of industry database vendors so that the JDBC would be as widely accepted as possible when it was ready for release. As can be seen from those vendors who have already endorsed the JDBC, it is sure to be widely accepted by the software industry.

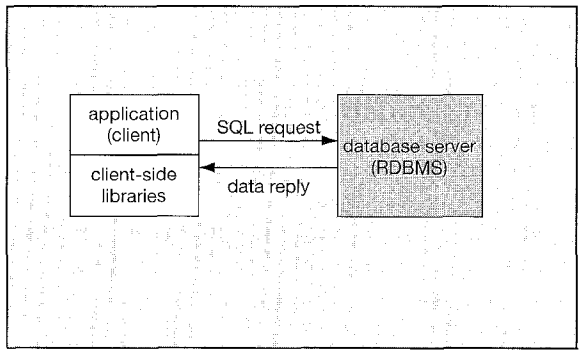


Fig. 1 Two-tier model for database access

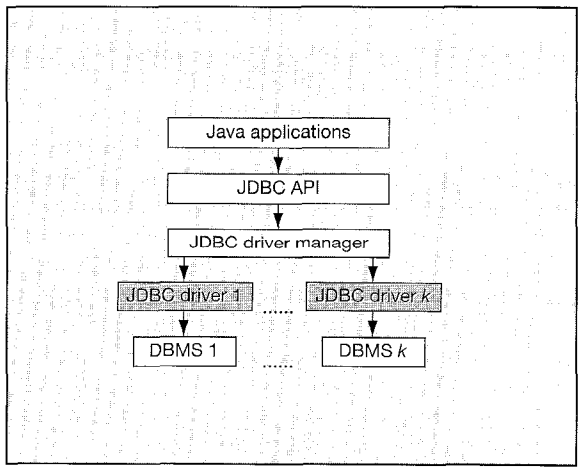


Fig. 2 General software architecture for the implementation of a two-tier model

The Structured Query Language (SQL) is an ANSI standard and all major commercial RDBMS vendors provide mechanisms for issuing SQL commands. It is typical for RDBMS and DBMS (database management system) to be used interchangeably because almost all major commercial databases are relational and support some form of SQL to allow the user to query the relations between data tables. So, the JDBC API must support SQL as it is. In fact, the JDBC standard is heavily based on the ANSI SQL-92 standard, which specifies that a JDBC driver should be SQL-92 entry-level compliant.

Using JDBC, it is easy to send SQL statements to virtually any relational database. In other words, with the JDBC API, it is not necessary to write one program to access a Sybase database, another program to access an Oracle database, another program to access an Informix database, and so on. One can write a single program using the JDBC API, and the program will be able to send SQL statements to the different kinds of databases. With an application written in Java and JDBC programming, one also does not have to worry about writing different applications to run on different platforms. The combination of Java and JDBC allows a programmer to write its code once and run it in different platforms.

The JDBC standard was created with the help of leading database and database-tool vendors such as Sybase, Oracle, Informix, Symantec and Intersolv, and it provides a programming-level interface for communicating with databases in a uniform manner similar in concept to Microsoft's Open Database Connectivity (ODBC), which has become the standard for accessing RDBMSs. The JDBC standard is in line with the requirement of the X/Open SQL Call Level Interface, the same basis as that of ODBC.

JDBC is a low-level interface, which means that it is used to invoke (or call) SQL commands directly. It is also designed as a base on which to build higher-level interfaces and tools. A higher-level interface can be implemented as user-friendly by using a more understandable or more convenient API that is translated behind the scenes into a low-level interface.

Technical approaches for accessing databases on the Internet

The JDBC API supports both two-tier and three-tier models for database access.[3] Four different technical approaches can be used to implement the two models. The two models and the software architectures for their implementation are investigated in this section.

Two-tier model

The two-tier model appeared with the advent of server technology. Communication-protocol development and extensive use of local and wide area networks allowed the database developer to create an application front end that accessed data through a connection (socket) to the back-end server. A two-tier model for

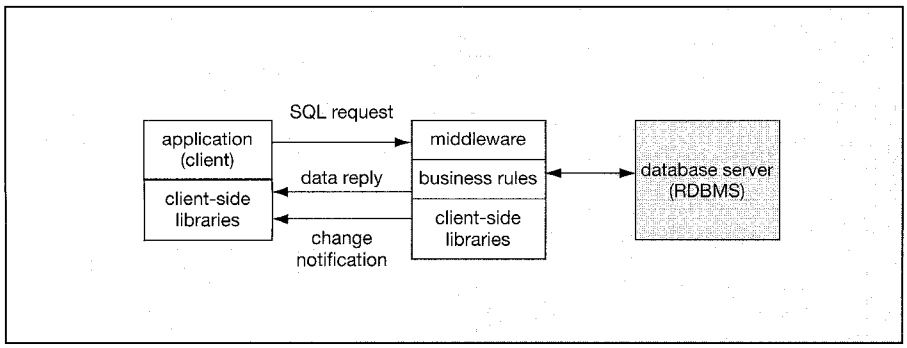


Fig. 3 Three-tier model for database access

the database access is illustrated in Fig. 1.

In the two-tier model, a Java applet or application talks directly to the database. This requires a JDBC driver that can communicate with the particular database management system being accessed. A user's SQL statements are delivered to the database, and the results of those statements are sent back to the user. The database 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 users' machine as the client, and the machine housing the database as the server. The network can be an Intranet, which connects employees or partners within a corporation, or it can be the Internet. Fig. 2 shows a general software architecture for a two-tier model, where the JDBC driver and the DBMS can be from different vendors.

The JDBC driver manager is used to open a connection to a database via a JDBC driver, which must register with the driver manager before the connection can be formed. When a connection is attempted, the driver manager chooses from a given list of available drivers to suit the explicit type of database connection. After a connection is formed, the calls to query and fetch results are made directly with the JDBC driver. The JDBC driver must implement the classes to process these functions for the specific databases.

Three-tier model

In the three-tier model, commands are sent to a 'middle tier' of services, which then sends SQL statements to the database. The database processes the SQL statements and sends the results back to the middle tier, which then sends them to the user, i.e. in a three-tier design, the client communicates with an intermediate server that provides a layer of abstraction from the RDBMS. The three-tier model is represented in Fig. 3.

A software architecture for a three-tier model with a JDBC-ODBC bridge is shown in Fig. 4, where the JDBC-ODBC bridge, ODBC driver and DBMS can be from different vendors.

JDBC-ODBC bridge

The software implementation of JDBC-ODBC bridge belongs to the three-tier model. As its name implies, the JDBC-ODBC bridge is a gateway that provides an interface from JDBC to ODBC drivers. Many database packages come with ODBC drivers, such as MS Access, FoxPro, dBase, Interbase, Sybase and Oracle.

The JDBC-ODBC bridge was developed by JavaSoft to take advantage of the large number of ODBC enabled data sources. As illustrated in Fig. 5, Java applets (or applications if a browser is not being used) are written by using the JDBC API. These JDBC calls are passed to the

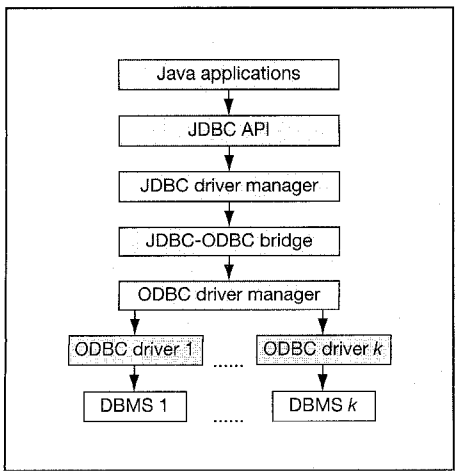


Fig. 4 Software architecture for the implementation of three-tier model with JDBC-ODBC bridge approach

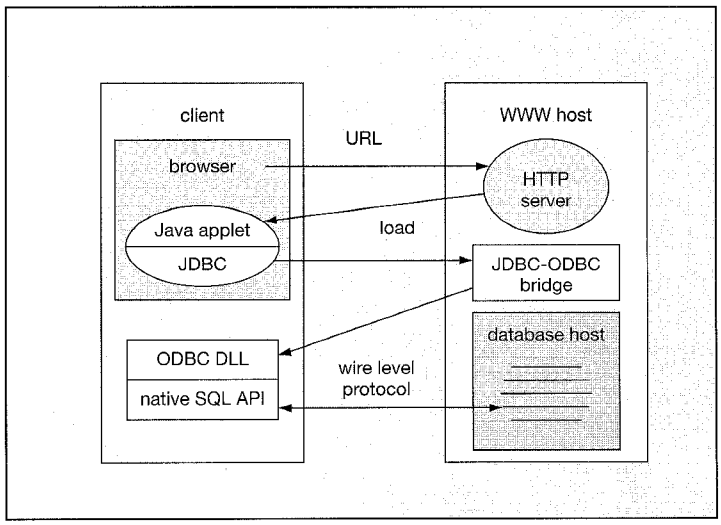


Fig. 5 Approach for using JDBC-ODBC bridge

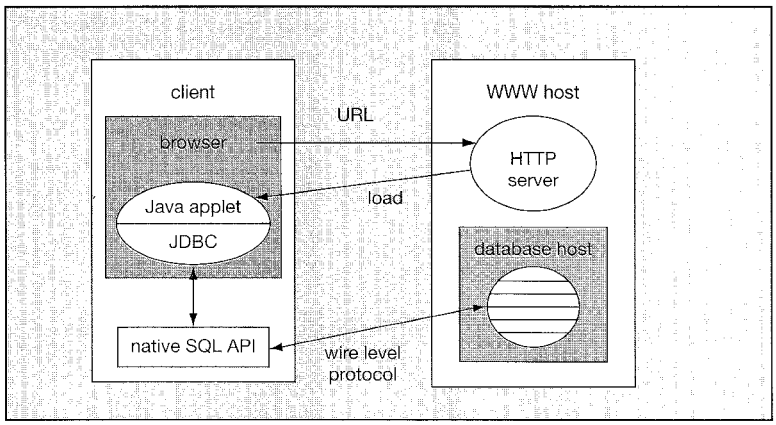


Fig. 6 Approach for using native-API partly-Java driver

JDBC-ODBC bridge and converted into ODBC APIs with C language. The ODBC calls are then passed to an appropriate ODBC driver for the back-end data store. The technical approach to software implementation for using the JDBC-ODBC bridge is illustrated in Fig. 5.

The primary advantage of using the JDBC-ODBC bridge is that because the JDBC calls are ultimately converted into ODBC calls, applications can easily access databases from multiple vendors by choosing an appropriate ODBC driver. However, if using this type of driver, the JDBC driver must be pre-installed on any client using the JDBC-ODBC bridge. The requirement to pre-install software implies the same type of software administrative burden as for traditional client/server applications.

Native-API and partly-Java driver

This JDBC implementation is a two-tier model and it connects the client to the database by using vendor-supplied libraries (e.g. client-library for a special vendor). These drivers are typically written in some combination of Java and C (or C++), as the driver must use a layer of C program in order to make calls to the vendor libraries (which are written in C). This kind of driver typically converts JDBC calls into the calls on the client API for Oracle, Sybase, Informix, DB2 or other DBMS. An illustration for the approach of the software implementation is shown in Fig. 6.

These drivers, like the JDBC-ODBC bridge, require that code (the vendor library) be installed on each client. Thus they have the same software maintenance problems as the JDBC-ODBC bridge.

Net-protocol and all-Java driver

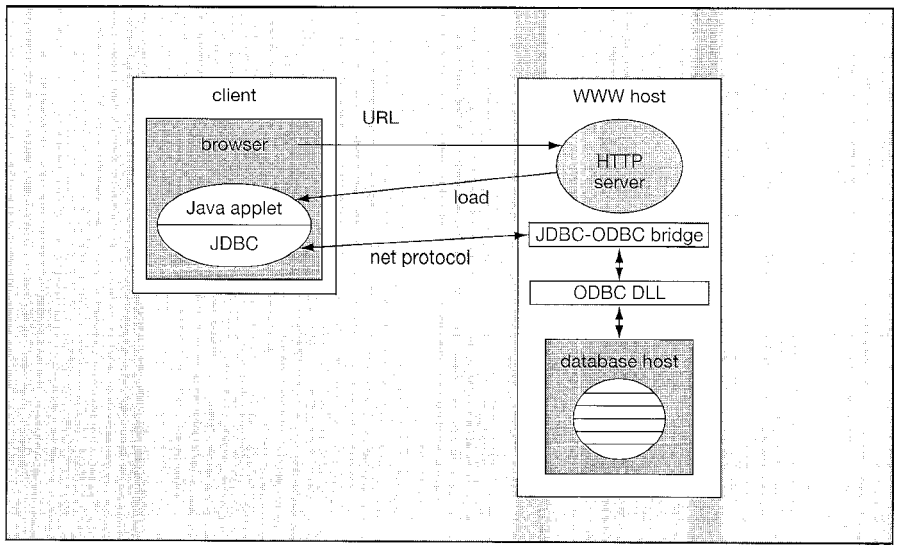
The software implementation of the JDBC driver translates JDBC calls into a database-independent net protocol, which is then translated into a database-specific protocol by a middle-tier server. This approach is shown in Fig. 7.

This type of JDBC driver can be written entirely in Java and can provide just-in-time delivery of Java applets. However, at a minimum, an additional communication hop along with protocol translation is required before the translation is made to ODBC. In addition, there is still the overhead of going from ODBC to the native protocol before the actual database is accessed.

Native-protocol and all-Java driver

This type of JDBC driver converts JDBC calls directly into the network protocol used by the specific database vendor. These drivers can be written entirely in Java and can provide for just-in-time delivery of applets. As these drivers translate JDBC directly into the native protocol, without the use of ODBC or native SQL APIs, they can provide for very high performance database access. One software implementation for the approach is shown in Fig. 8.

Fig. 7 Approach for using net-protocol all-Java driver



In this kind of technical approach, the Java applets with JDBC driver are downloaded into a browser without any prior client software installation.

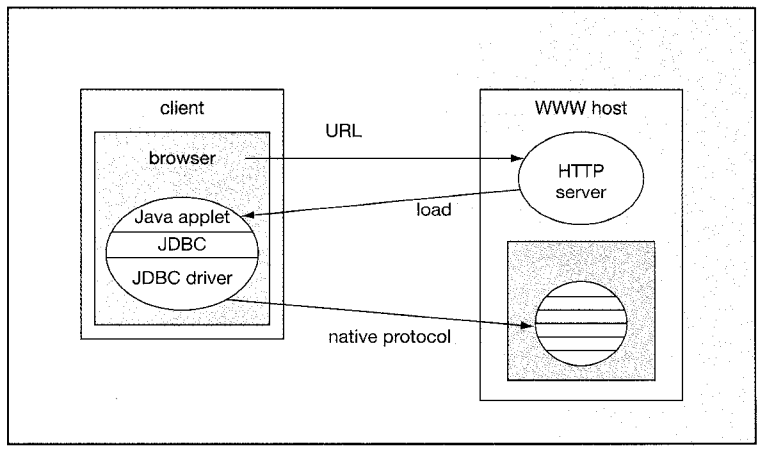


Fig. 8 Approach for using native-protocol all-Java driver

Design and implementation for accessing databases on the Internet by using the JDBC-ODBC bridge approach

ODBC has already established itself as an industrial standard; what better way to make JDBC usable by a large community of developers than to provide a bridge between the JDBC driver and ODBC driver. JavaSoft turned to Intersolv to develop a bridge between the two drivers, and the resulting JDBC driver and the JDBC-ODBC bridge is now included with the Java Developer's kit.

A design and implementation of Java software for database access on the Internet is proposed in this section; the software is implemented by using the JDBC-ODBC bridge approach mentioned in the subsection on the 'JDBC-ODBC bridge'.

Design of the Java software

The Java software is designed and implemented as an applet with interactive query operations, which can accomplish a number of tasks, mainly: to provide a user-friendly interface on a browser; to allow users to enter the name of the database wanted; to connect to a database server on the Internet; to wait for users to enter an SQL query passed to the database server; and to display the results of the query in an output area of the browser.

The development of the software is based on an event-driven design scheme. The kernel component of the software is an interactive applet. Fig. 9 shows the flow diagram of the interactive applet for database access on the Internet.

The structure of the Java applet has a well-defined flow. In the initialisation phase, the first thing is to define the various objects and their constructs, then design and layout an interactive user interface. Next, it is necessary to design an event handler which is used to signal the applet when users have entered some information, for

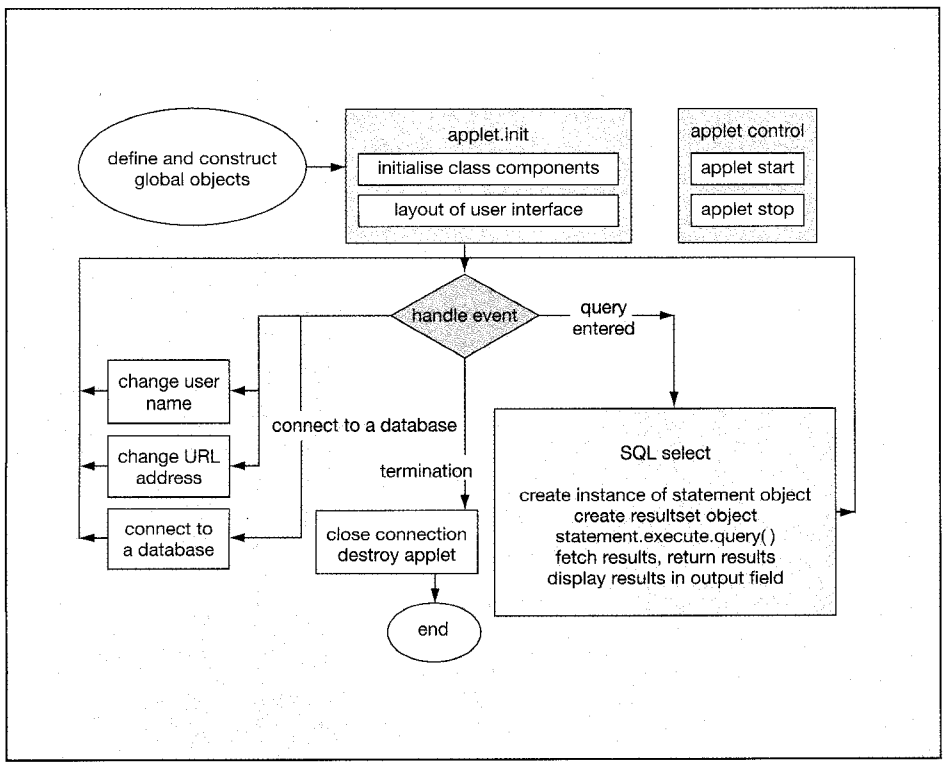


Fig. 9 Flow diagram of interactive applet for database access on the Internet

instance, connecting to a database, or executing an SQL query, and so on. Depending on the contents that the users enter, the event handler selects a suitable method to process them. Finally, we should close all the connections to the data source before the applet is terminated. The technical detail for the implementation of the applet is described in the next subsection.

Technical detail of JDBC classes and implementation of the software

The interactive applet is mainly made up of five parts; the first part is the initialisation phase; the second is the layout of user interface; the third is the event handler; the fourth is to connect to a database, to process SQL query operation, and to fetch and to display the results; and the last one is to close all the connections and terminate the applet. The first part and the last part are normal designs for a Java applet. In the second part, we use GridBagLayout, a Java layout manager, to position the components of the user interface in the applet window on the browser. GridBagLayout is flexible and offers a quick way of producing an attractive interface. The core contents of the applet for accessing a database are the third and the fourth parts, i.e. the *event handler phase* and the *connection and SQL access* phase which are investigated in detail in the following:

• *Event handler phase:* The event handler phase is designed and implemented to watch four kinds of events: the user presses the *enter* key in *Database URL* field (*DBurl*) which includes the protocol for the JDBC driver and the name of the data source mapped to a real database, *User Name* field (*NameField*) which can be entered as a user name for a security check, *SQL Query* field (*QueryFieldTextAreas*) which should be entered as a query command for accessing the database, and the user clicks on the 'connect to database' button. The outline code for the event handler phase is provided as follows, which is contained in a generic *handleEvent* method:

```
public boolean handleEvent(Event evt) {
// Process it when the enter key pressed in User Name
field.
 if (evt.target == NameField)
   {char c=(char)evt.key;
   if (c == '\n')
      { Name=NameField.getText( );
        return true;
      }
    else { return false; }
   }
// Process it when the enter key pressed in Database
URL field.
 if (evt.target == DBurl)
   {char c=(char)evt.key;
   if (c == '\n')
      { url=DBurl.getText( );
        return true;
      }
    else { return false; }
   }
// Process it when the enter key pressed in SQL Query
field.
if (evt.target == QueryField)
   {char c=(char)evt.key;
   if (c == '\n')
      {
       // Call SqlSelect method to query the database
with using entered SQL statement, and return the result
in Query Result field on the interface.
       OutputField.setText(SqlSelect(QueryField.getText
( )));
       return true;
      }
    else { return false; }
   }
// Process it when the user clicks on the "Connect to
Database" button.
if (evt.target == ConnectBtn)
   {
   url=DBurl.getText( );
   Name=NameField.getText( );
   try {
    // Register a JDBC-ODBC bridge.
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    // Open a database connection with using the entered
Database URL and User Name.
    con = DriverManager.getConnection(url, Name, " ");
    .....
    ConnectBtn.setLabel("Reconnect to Database");
   }
   catch( Exception e ) {
    e.printStackTrace( );
    OutputField.setText(e.getMessage( )); // Error
message is shown in OutputField.
   }
 return true;
 }
return false;
} // The end of handle Event( ).
```

The standard format for this method includes the various *event* classes where all the properties are set. As the enter key is pressed in the *User Name*, *Database URL*, and *SQL Query* fields on the interface, it looks for these events and sets an appropriate global variable in each of these fields, then processes them. When a SQL statement is entered in *SQL Query* field, it gets the contents and passes them to the *SqlSelect* method. The *SqlSelect* method executes a SQL operation according to the entered SQL statement and returns the result. These results are shown in an output field, *Query Result*, on the interface. When the 'connect to database' button on the interface is pushed, it connects to a database

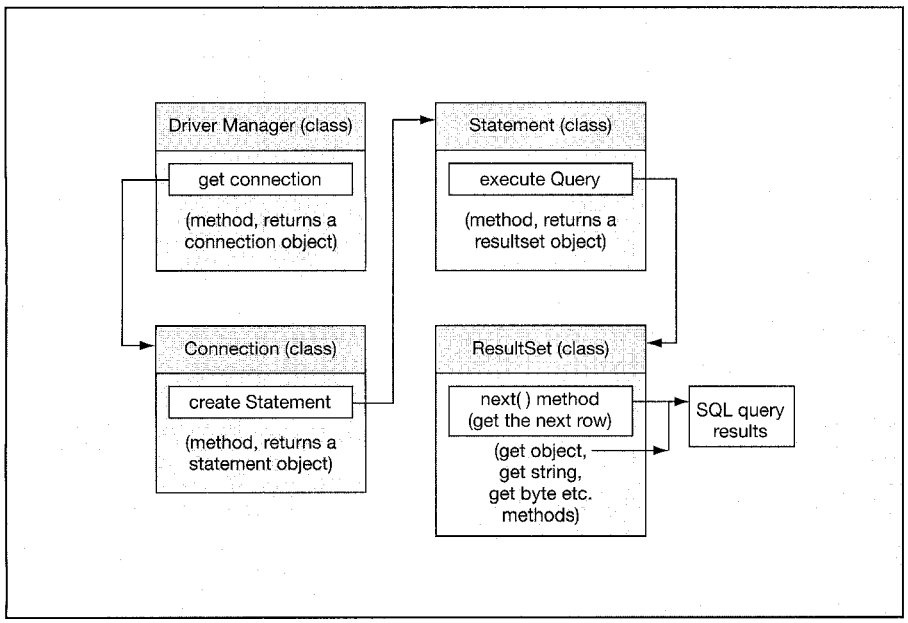


Fig. 10 Diagram relating the four JDBC classes used in the Java program

and processes the user's query operation with *SqlSelect* method. The process, together with SQL operations, is discussed in the following:

• *Connection and SQL access phase:* Before we discuss the implementation of the connection, SQL operation and display of the results, it is necessary to investigate the four JDBC classes used in our Java program. Fig. 10 relates the four classes.

Fig. 10 is a skeleton of the main JDBC classes used for connecting and accessing to a database. It shows the relationship between these classes. The Figure does not list all the methods available in the respective classes. The list of the complete classes and methods is referred to in the documentation for Java JDK1.1.1).

The procedure for opening and connecting to a database is provided in the 'if(evt.target == ConnectBtn) { .....}' part shown in the above *Event handler phase*. If the user clicks the 'connect to database' button, it is connected to the data source specified in the 'url' and 'name' variables which represent a suitable database and the user name satisfied by the security, respectively. The sentence Class.forName('sun.jdbc.odbc.JdbcOdbcDriver') creates and registers an instance of the JDBC driver wanted; here we make the *JDBC DriverManager* choose the sun.jdbc.odbc.JdbcOdbcDriver to connect to the database source. The sentence 'con = DriverManager. getConnection(url, Name, " ")' actually makes the connection by using the variables 'url' and 'Name' which represent the contents entered in the *Database URL* and *User Name* fields. The procedure for the SQL operation, the display of the results and the closing of the connection is implemented as a method named *SqlSelect*, and the outline code for the method is provided as follows:

```
// Method SqlSelect: implement a SQL access to a
database.
public String SqlSelect(String QueryLine) {
The definitions of variables, ......;
  try {
    Statement stmt = con.createStatement( );
    ResultSet rs = stmt.executeQuery(QueryLine);
    columns=(rs.getMetaData( )).getColumnCount( );
    The console control, ......;
    while(rs.next( )) {
     for( pos=1; pos<=columns; pos++) {
     // The contents of each column of the database can
be output correctly.
       ......;
       Output+=rs.getObject(pos)+"\t ";
      }
      Output+="\n";
    }

   // Close the ResultSet connection.
   rs.close();
   // Close the Statement connection, that is to close
input-output query connection streams.
    stms.close( );
  }
  catch( Exception e ) {
   e.printStackTrace();
   Output=e.getMessage(); // Error message is put into
Output area.
  }
 return Output;
} // The end of SqlSelect method.

// Method destroy( ): terminate the connection.
public void destroy( ) {
 // Close the connection to the database, that is to
terminate the connection.
```

```
 try {con.close();}
 catch( Exception e ) {
    e.printStackTrace();
   System.out.println(e.getMessage());
  }
} // The end of destroy method.
```

Firstly, the sentence 'Statement stmt = con.create Statement()' creates a *Statement* class that is required to execute a query operation. The *Statement* class results from the *createStatement* method in the *Connection* class (see Fig. 10), which makes the 'stmt' instance linked to an actual connection to the data source. Then, the sentence 'ResultSet rs = stmt.executeQuery(QueryLine)' in turn is linked to the data source via the *Statement* class. The *Statement* class is used to retrieve the information from the data source. It contains the *executeQuery* method which implements the SQL access operation and returns a *ResultSet* class. The *ResultSet* class is used to get the query results from the data source via the connection. The 'while(rs.next()){ ..... }' part in the *SqlSelect* method is implemented to get and display all results of the query operation. The *next* method of the instance *rs* of *ResultSet* class is used to fetch each row one by one. A general method, *getObject*, is used for getting a query result column by column. It will attempt to fetch the result in the form of its assignee and put the result into the output field for displaying. After the query operation is finished, the connections of the *ResultSet* and the *Statement* classes are closed in the *SqlSelect* method. Lastly, the *destroy*( ) method is executed to terminate the connection to the data source.

In addition, the mechanism for catching any exceptions is considered throughout the whole program. The '*try* { main process ...} *catch*(Exception e) { exception process ...}' programming technique is used in the software so that any exceptions will be tripped while the information in the data source is being queried, retrieved and displayed. This mechanism ensures that the exceptions will be caught and the error message will be printed to the console if any exceptions occur during the program run.

Instances for the execution of the software

Our prototype software can be run on a platform with Netscape Communicator 4.0 browser and any browsers for supporting the Java JDK 1.1 or over. Since the design scheme and implementation of the Java software follows the technical approach which uses a JDBC-ODBC bridge (see earlier), the Java JDBC driver and JDBC-ODBC bridge should be installed at each client machine. Any users who have been authorised, whether they are at the remote machine or in a local network, can access the database by using the SQL sentences of ANSI 92 standard.

Readers who are interested in examples and illustrations of test results should contact the authors.

Summary

Java is a 'write once, run anywhere' language. This means that Java programs may be deployed without changing any of the computer architectures and operating systems that run the Java virtual machine. For large corporations, just having a common development platform is a big saving. No longer are programmers required to write different codes to satisfy the different platforms that a large corporation may have. All of the leading OS and browser platforms are building Java virtual machines into their systems. Additionally, major development tool vendors and application suppliers are providing support for Java.

The database is used to collect and manipulate data and it forms the foundation of the information infrastructure in the world. Using the Internet to access the various databases is an important research direction. Java offers several benefits to the developer creating a front-end application for a database server. Various technical approaches for supporting the database access on the Internet have been proposed and investigated in this article. The design and implementation of a Java JDBC software is proposed and discussed in detail. Our Java software provides an interactive and user-friendly interface on the browser which supports Java JDK 1.1 or over and the software has the capability for connection, SQL access, retrieval and displaying of a database on the Internet.

Java, being robust, secure, easy to use, easy to understand, and automatically downloadable on a network, is also an excellent language basis for database applications. What is needed is a way for Java applications to talk to a variety of different databases. JDBC API provides the foundation of the mechanism for doing this. Further research should be to develop a general software or tool which can be used to access flexibly different vendors' databases on the Internet.

References

1 GOSLING, J., JOY, B., and STEELE, G.: 'The Java language specification' (Addison-Wesley, Menlo Park, Calif., 1996)
2 URQUHART, K.: 'Java's open future', *IEEE Micro*, 1997, **17**, (13), pp.11–13
3 HELLER, P., and ROBERTS, S.: 'Java 1.1 developer's handbook' (SYBEX Inc., Alameda, 1997)
4 JENNINGS, R.: 'Choose the right data access tool', *Visual Basic Programmer's Journal*, 1997, **7**, (8), pp.22–24
5 JEPSON, B.: 'Java database programming' (John Wiley & Sons Inc., New-York, USA, 1997)
6 LINDHOLM, T., and YELLIN, F.: 'The Java virtual machine specification' (Addison-Wesley, Mass., 1997)

Huiwei Guan is with the Department of Computer Engineering, Shanghai University, Shanghai 200072, Peoples' Republic of China and is also with the GMD-IPSI, Darmstadt, Germany. Horace Ip is with the Department of Computer Science, City University of Hong Kong, Hong Kong. Yanchun Zhang is with the Department of Mathematics & Computing, University of Southern Queensland, Australia.