Applications of JAVA programming language to database management

Bradley F. Burton and Victor W. Marek*

Department of Computer Science
University of Kentucky
Lexington, KY 40506-0046
e-mail: {bfburton|marek}@cs.uky.edu

1 Motivation

The Java programming language [1,3] from its inception has been publicized as a web programming language. Many programmers have developed simple applications such as games, clocks, news tickers and stock tickers in order to create informative, innovative web sites. However, it is important to note that the Java programming language possesses much more capability. The language components and constructs originally designed to enhance the functionality of Java as a web-based programming language can be utilized in a broader extent. Java provides a developer with the tools allowing for the creation of innovative network, database, and Graphical User Interface (GUI) applications. In fact, Java and its associated technologies such as JDBC API [11,5], JDBC drivers [2,12], threading [10], and AWT provide the programmer with the much-needed assistance for the development of platform-independent database-independent interfaces. Thus, it is possible to build a graphical database interface capable of connecting and querying distributed databases [13,14].

Here are components that are important for building the database interface we have in mind.

- JDBC – The Java Database Connectivity module is a standardized database access solution (JDBC is packaged in Java 1.1 and later versions).
- Java Threads – The Java programming language is multithreaded language. It provides multiple threads, light-weight quasi-processes, handling different tasks at the same time.
- AWT – The abstract window toolkit provides the ability to build platform-independent graphical user interfaces.

The combination of the above technologies provides the fundamental building blocks that we use to develop a distributed database interface. This package, Java Distributed Query Dispatcher (Java DQD), is a platform-independent GUI application capable of querying multiple heterogeneous databases simultaneously. JavaDQD is based upon the following ideas:

1. JavaDQD application establishes database connectivity using JDBC API and drivers. By utilizing the JDBC API, JavaDQD provides uniform access to any database providing a JDBC driver. That is, a user may connect to any database for which a JDBC driver is provided. For our purposes, we will connect to Sybase and MiniSQL using FastForward and mSQL-JDBC respectively. The mSQL-JDBC driver, we use, has been modified to implement the desired DatabaseMetaData functionality. As other JDBC drivers are developed and become available, JavaDQD will allow for their incorporation. Thus, JavaDQD is an extendable tool for multiple database connectivity.

2. JavaDQD interface uses Java threads to allow a user to connect to multiple databases simultaneously. The user can submit query distributed over multiple databases; hence, when a user requests data from multiple sources, a Java thread is started for each database that needs to be queried. Each thread accepts a query string, queries a database, and accepts the result. Thus, many databases can be queried at virtually the same time.

3. JavaDQD uses the AWT and third party GUI components in providing the capability to present the

*Corresponding author.

user with a QBE-like interface. The QBE-like interface creates a single virtual environment displaying database information about all current connections. In essence, the user benefits from a single interface that can query across multiple databases as if the multiple databases were one.

Therefore, the technologies typically considered to enhance Java's web-based ability provide a basis for constructing a versatile distributed database interface. JavaDQD application utilizes Java's AWT and third party classes to present to the user a QBE-like graphical interface to query and manipulate a virtual database consisting of multiple heterogeneous databases unified by JDBC and Java threads. The following sections discuss the implementation methodology and observed results of JavaDQD.

2 Implementation

The JavaDQD application, that we have developed, incorporates Java and JDBC to manage distributed database querying and to provide the user a user-friendly environment to create distributed queries. We will mention the key ideas or methodology supporting the implementation of JavaDQD. Namely, we will explain the key Java and JDBC concepts used, our previous work with Java and JDBC, and our distributed querying approach.

2.1 Methodology, Java and JDBC

The Java Development Kit (version 1.1), released in the Spring of 1997, was used in the development of the Java code produced in the project. This version of the JDK does provide (like the previous version) a Java interpreter and compiler, but most importantly, the JDK 1.1 provides the Java Database Connectivity module called JDBC.

The Java programming language developed by SunSoft is an excellent programming language. The Java language may be described as relatively simple, object-oriented, distributed, and portable. However, one of the more important capabilities provided in the Java language is its capability of producing platform-independent programs. The Java language provides an excellent framework for network-aware programs that can run on any platform that has a Java interpreter. A Java program may be developed to run as an applet, a program that is downloaded over the Internet and run on the client, or as an application, a program that resides on the client side. In either case, the Java programmer has the ability using built-in classes and methods to access and use remote web space data such as text, images, or sound in their programs. Similar to the network capabilities just mentioned, the recent addition of JDBC allows a Java programmer to connect and query remote databases using an API supplied in the JDK. Thus, Java can be viewed as an excellent database programming language because of platform-independence, network-awareness, and JDBC.

The development and inclusion of JDBC does extend the Java programming language capability of network programming. JDBC is a package that has been recently added to the JDK. JDBC offers a generic SQL database framework that defines a uniform API for a variety of data sources. Actually, the JDBC API is a package of abstract classes that must be defined for specific database sources. This implies that JDBC can be viewed from a high-level abstract view or from a low-level database specific view. A high-level, application programmer view of JDBC is an API, called the JDBC API, that provides methods allowing an application to connect, query and manipulate multiple databases. By contrast, the low-level, database specific view of JDBC interprets it as a package of abstract classes that must be implemented for specific databases. That is, a database specific implementation of the JDBC abstract classes, called a JDBC driver, must be provided in order for the Java database programmer to access the database. Once a programmer implements a JDBC driver for a particular source, the driver becomes an abstract SQL engine whose details are internalized and can be accessed through the high-level JDBC API. Consequently, a database application obtaining database access through the JDBC API will work with any data source providing a JDBC driver. For instance, if a programmer developed an application powered by the JDBC API, the application would be able to connect to any data source providing a JDBC driver, be it Sybase, MiniSQL, or Access. Thus, JDBC is a powerful, flexible database connectivity module for the Java programming language.

2.2 Previous Work

Prior to developing JavaDQD, we researched Java and JDBC in order to develop a platform-independent database-independent database interface named JavaQD (Java Query Dispatcher). It is a Java client-side application that allows a user to query multiple databases by the way of SQL-like interfaces or QBE-like interfaces. JavaQD was constructed using the Java language and established database connectivity using JDBC API and JDBC drivers. Most importantly, the QBE-like interface utilized JDBC drivers' capability to probe a database's metadata. As a result, JavaQD's QBE-like interfaces were built dynamically reflecting information about the current database connection such as the available tables, available columns, or available data types. Consequently, JavaQD provided multiple database access and a user-friendly query interface.

The JavaQD application did demonstrate the ability to utilize Java and JDBC to build an effective user interface for querying many database engines. However, JavaQD could only manage one database connection at a time. Similar to our JavaQD, non-Java systems for querying remote databases have been proposed. In particular, the technique for using URLs of remote databases for connection has been introduced by Konopnicki and Shmueli in [7]. Their proposed interface, just as JavaQD, does not allow for simultaneous connection to several remote databases. Our approach solves this problem. Specifically, we realized JavaQD could be extended to handle multiple simultaneous connections; and, therefore, JavaQD could be migrated to a distributed database interface, which we call JavaDQD. That is, JavaDQD could query multiple heterogeneous databases simultaneously. Thus, the idea for JavaDQD was formed. JavaQD would be modified to handle distributed databases by utilizing Java threads and handling pre- and post-processing of queries.

2.3 Distributed Approach

As stated above, the JavaDQD application was modified from an earlier work JavaQD in order to handle distributed query processing. Primarily, JavaDQD implements distributed querying through Java threads and localized pre- and post-processing. We will outline the methodology of the distributed querying in JavaDQD.

1. *Pre-process a user's query to create query strings.* Query strings needed to query each of the distributed databases must be determined. These query strings will obtain data from multiple databases that ultimately might be included in the final query result. Similarly, a *collection* query string must be constructed to query all the distributed results.
2. *Fork a thread to query each of the distributed databases that need to be queried.* Each thread uses its given *distributed* query string to query the appropriate database, gather the result, and then place the result in a temporary database. The temporary database is a database used to store the results of all the databases queried. The tables within the temporary database will later be queried using the *collection* query in order to obtain the final result. Specifically, each thread gathers the result from the *distributed* query, creates a table in the temporary database to store the result, and then, the thread populates or stores the result in the table. It is important to realize a thread, by utilizing the ResultSetMetaData methods in order to determine data types and column sizes for the new table, creates a table within the temporary database.
3. *Wait for each thread to finish.*
4. *Query the temporary database with the* collection *query.* The temporary database is queried and the result of the query is displayed to the user in a window.

The distributed query processing is handled internally by pre- and post- processing. Thus, a user of JavaDQD is not required to know information about the individual database connection. The utilization of the JDBC API and JDBC drivers allows the distributed nature of queries be transparent to the user. Consequently, the JavaDQD interface presents a single virtual database of distributed databases.

Security issues for JavaDQD are handled by the JDBC driver. Our application calls the JDBC method getConnection(). This method facilitates the connection to the remote server. In particular this implies that the JDBC driver is responsible for a proper handling of security issues. Thus a user must investigate the security of JDBC drivers called by JavaDQD since this affects the JavaDQD security.

3 User-interface

We have described the methodology of the development of JavaDQD. However, now, we will address JavaDQD from a user's perspective. JavaDQD's user interface provides a user with the capability to build distributed queries in a graphical environment organizing multiple heterogeneous databases as if they are one. It is important to note that JavaDQD's user-interface does not require the user to know the schema of the each connected database; JavaDQD probes each connected database's metadata in order to determine knowledge of the database and present the user with an informative interface. Thus, the user has the ability to query across multiple databases with the ease and flexibility accompanied with querying only one database. In this section, we will give details of specific elements of the JavaDQD user-interface such as the *connection dialog*, *create* interface, *select* interface, and *insert* interface.

JavaDQD's database connection dialog provides the user with the ability to connect to local or remote distributed databases and the ability to connect to a temporary database. In order to make a database connection, a user must provide a database URL and, if necessary, a user name and password. A database URL specifies the JDBC driver, the database source, and the database port. The database URL pictured in the connection dialog below allows the user to attempt to make a connection to a miniSQL database server on the machine, shelley.ca.uky.edu. For instance, the URL specifies the miniSQL database server is listening to port 1114 and the connection should make wxdb (the current remote database) the working database. Moreover, the database connection dialog allows the user to enter a user name and password if required. For simplicity, the collection of an URL and user name can be saved in the configuration file that is loaded when JavaDQD is initiated; thus, the user can save information about frequently accessed databases.

Once a database connection is made, the user can begin to query the database using dynamically built QBE interfaces.

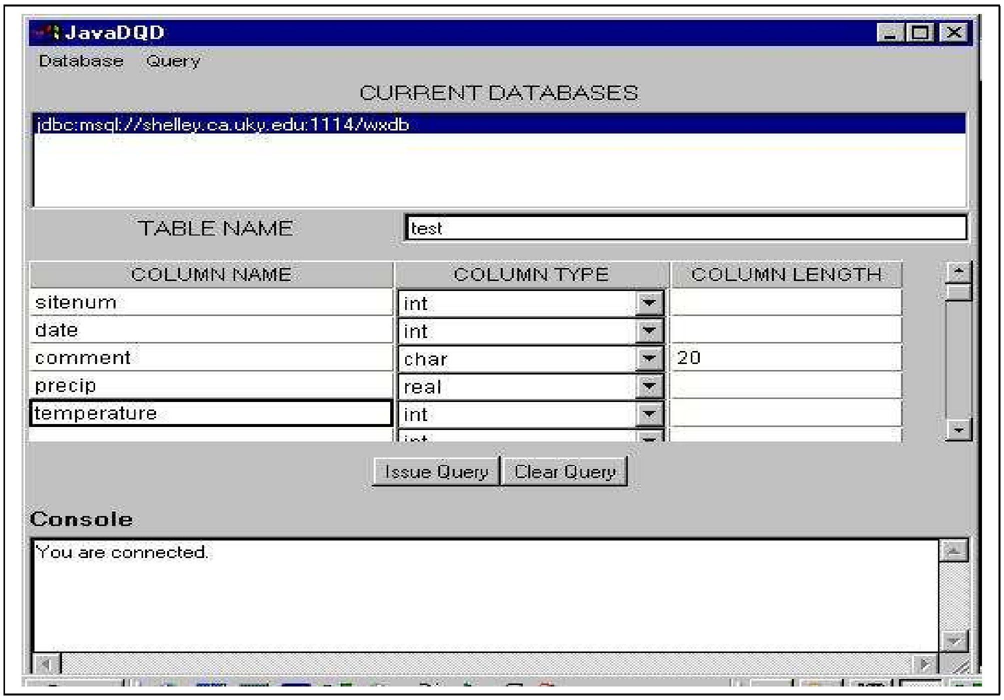
The JavaDQD's QBE-like distributed database interface demonstrates a user-friendly interface that utilizes the knowledge of the database schema. The QBE-like interfaces incorporate the GUI capabilities of Java and MCT, Microline Component Toolkit, such as radio buttons, lists, buttons, grids, and scrollbars. These capabilities are used to construct interfaces derived from the QBE specification proposed by Zloof [15] and utilized by IBM. Indeed, the GUI components and the ability packaged in the JDBC API to probe the schema of a database allows the QBE-like interfaces to be built dynamically after a connection to a database is established. Therefore, the QBE-like interfaces do show the database-independent ability of JDBC. The QBE-like interfaces provided in the JavaDQD application are create, drop, select, insert, update, and delete [1]. We will detail the *create*, *insert*, and *select* interfaces.

The QBE *create* interface provides the user with the ability to create a new table in a database. The *create* interface has three main components. First, the *create* interface provides a selection list that provides a list of all the current databases. The user must select a database in which the new table will created. Second, the *create* interface provides a text field to enter the new table name. Third, the interface provides a grid in which one can define the columns of the table. One can define the column name, pick a supported data type, and a column length. The supported data types are determined dynamically by probing the schema of the selected database. The QBE *create* interface is shown below.

The QBE *select* interface allows the user to select data from the distributed database. This data is subsequently presented in a result frame. The QBE select interface presents a list of available tables from which data can be selected. The list of available includes all tables from all current database connections. One should note that we choose to identify tables by database URL and table name (refer to sample *select* interface below). Once a user chooses a table, a select grid is constructed to designate the criteria the result data must adhere to. The user can specify which columns will be displayed by the checking a column in the *view* row. Also, a user can specify an example of what a returned row of data should be. The user can express a column value to be $=, <, >, \leq, \geq$, LIKE, or $<>$ a literal value; a literal value is a character string or a numeric value. The expression criteria placed in one row constitute a conjunctive clause, and criteria placed in separate rows constitute a disjunction of conjunctive clauses. Lastly, once a user chooses two or more tables from the available list, a *join item* row is present. The *join item* row provides a series of radio buttons to specify the columns to join tables. One can select one column from each table, and the selected columns signify equal column values. Thus, one does have the ability to join multiple tables on one column value. Admittedly, this is a limitation; however, using the concept of radio buttons and with additional *join item* rows, one can construct a conjunction of join items and achieve full join functionality.

The QBE *insert* interface provides the user with the ability to insert a new row into a table. The *insert* interface presents a list of available tables to insert into.

[1] Space restrictions permit us to show only few screens.



Create interface

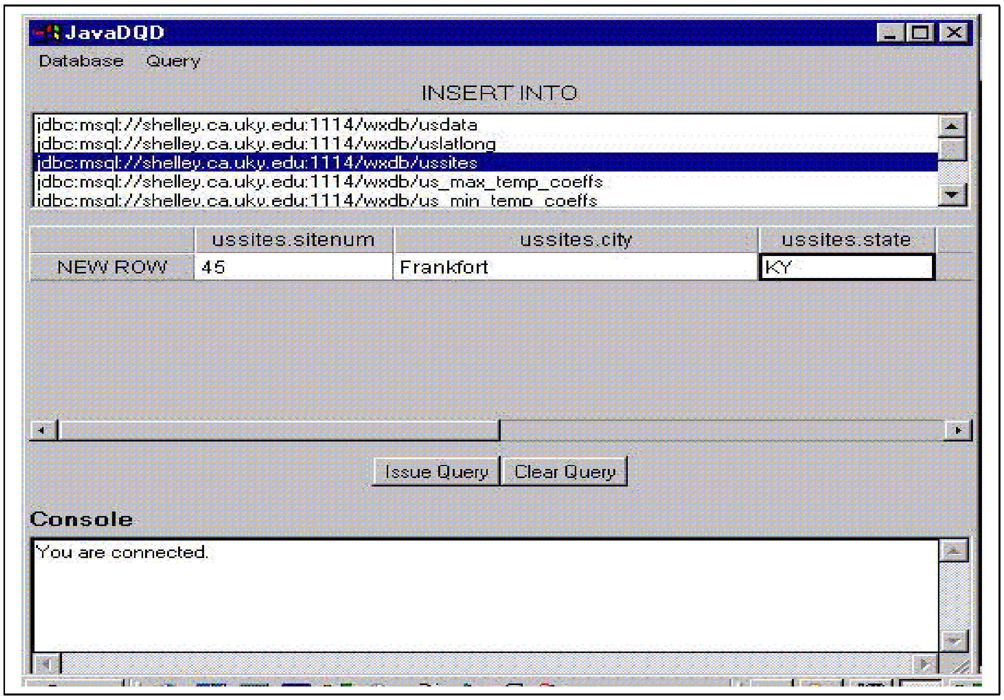
Again, the list of tables includes all tables from all current database connections, and the tables are identified by database URL and table name. Once a table is chosen, a grid is displayed with the columns and cells to enter the new data. The QBE *insert* interface is displayed below.

As should be clear from the above discussion, JavaDQD's interface provides the user with a friendly, informative interface that can be used with many data sources. The QBE-like interfaces, most importantly, demonstrate the database-independent ability of the Java programming language and the JDBC API. The project demonstrates that the JDBC API can be used to generate platform-independent distributed database interfaces. It is clear that Java GUI capabilities can be used to build better database interfaces. The consequences of the Java and JDBC combination allow for further development and will be a subject of database research in years to come.

4 Results

The work described above on the JavaDQD application demonstrates the feasibility of using Java and JDBC in developing distributed database interfaces. In our final remarks, we will critically consider the accomplishments and restrictions of JavaDQD as well as the current state and future of the technologies utilized in the development of JavaDQD. Several conclusions drawn from the development of JavaDQD are JavaDQD is a prototype distributed database interface, Java threads provided an efficient mechanism in distributed query processing, and Java's AWT and third party classes were useful in developing a user-friendly QBE-like database interface. The conclusions are presented below:

- JavaDQD database interface should be characterized as a prototype Java distributed database interface. JavaDQD has been successfully tested querying remote MiniSQL databases located on different machines using the modified mSQL-JDBC driver. It is important to note since the JavaDQD interface handles database connectivity using the JDBC API. The test does signify that JavaDQD can be utilized to query any database providing a JDBC driver. That is, JavaDQD can query Sybase, Oracle, Informix, and other databases providing a JDBC driver that implements the minimal set of classes and methods used by JavaDQD. Thus, JavaDQD is an extendable dis-



Insert interface

tributed database interface.

- JavaDQD effectively queries multiple databases by Java threads. The JavaDQD application does handle the network connections and query processing by the way of Java threads. The Java threads, when given a query string, submit the query to the desired database and collect the results. By handling the multiple queries in threads, one achieves an advantage of querying multiple databases virtually simultaneously not sequentially. Therefore, JavaDQD's utilization of threads greatly enhance the performance of distributed database processing.

- The combination of JDBC, Java threads, AWT, and third-party classes facilitates the ability to develop a QBE-like interface presenting a virtual database. The interface presented to the user transparently organizes the multiple connections utilizing database metadata information to present the user a friendly, informative interface. JDBC DatabaseMetaData class is used to query each connection's available tables, columns, and data types. The resulting metadata is used to establish an environment displaying multiple heterogeneous databases as if they were one. Consequently, the user can query multiple databases just as he or she queries one database. Thus, the JavaDQD interface can be used to query multiple databases located on different machines and stored in different database engines.

Just as we stated some conclusions above, we also need to state restrictions involved in using JavaDQD to query distributed databases. The JDBC drivers and temporary databases utilized in conjunction with JavaDQD must be considered carefully. The restrictions are stated below.

- JDBC drivers used with JavaDQD must implement a minimal set of classes and methods. The JavaDQD application relies on JDBC classes and methods implemented in a JDBC driver for database connectivity; thus, in order for JavaDQD to connect to a particular database engine, a JDBC driver must exist for the database implementing all classes and methods utilized by JavaDQD. In particular, JDBC drivers accessed by the JavaDQD application must implement at least the following classes and methods:

- The temporary database must be the *greatest*

Classes	*Methods*
Connection	close(), getMetaData(), createStatement(), getConnection()
Statement	execute(), executeUpdate()
DatabaseMetaData	getTables(), getColumns(), getTypeInfo()
ResultSet	getMetaData(), getString(), next()
ResultSetMetaData	getColumnCount(), getColumnDisplaySize(), getColumnTypeName(), getTableName(), getColumnName()

common denominator of all databases involved in a distributed select query. That is, all the possible data types from all databases involved in the query must be available in the system serving as a basis for temporary storage. The temporary database is managed by JavaDQD to store results of each distributed database queried and to obtain the final query result. Since the temporary database must store the results from all distributed database queries, it must provide all the data types that are possible in the results. Therefore, the temporary database should be the greatest common denominator of the distributed databases queried.

Similarly, a JDBC driver (if fully compliant) allows the user to submit the queries in SQL-92 standard. This does *not* imply that the remote server must support the SQL-92 natively. What is implied is that the JDBC driver must implement a mechanism for conversion of SQL-92 queries into the remote database query language. In particular, we are trusting the JDBC driver for each particular remote database to be correctly implemented (which is not always the case, see conclusions below). Moreover, JDBC does not support types beyond SQL-92 standard. Thus, even if the remote database supports additional types JavaDQD will not be able to handle these types.

In addition to reporting the specific results and facts, the research and implementation of JavaDQD lead us to draw several conclusions about the current state and future of Java and JDBC technologies.

1. The JDBC API, currently, is a versatile interface for distributed database connectivity. Java facilitates platform-independence in addition to the JDBC API being a standardized set of abstract methods, which define database access in Java. Thus, by using Java and JDBC, JavaDQD is a platform-independent application that possesses the ability to connect, query, and manipulate any database in, which provides a JDBC driver implementing the minimal set of methods needed by JDBC.

2. The reality, however, of the current state of JDBC drivers is not satisfactory. Many JDBC drivers are in an immature stage. That is, many are not JDBC-compliant or do not implement the minimal set needed by JavaDQD. However, one must realize the immature state of JDBC drivers did not hinder the development of JavaDQD; it did restrict the ability to test on multiple database engines.

3. Java's AWT and other third party GUI classes are useful in developing user-friendly platform-independent interfaces. We conclude that JDBC, the AWT, and third-party GUI classes provided an excellent base for developing JavaDQD.

The future potential of Java distributed database interfaces is very promising. However, in order for that potential to be realized, several steps need to be taken by database vendors and developers. First, the capability of the JDBC standard must be recognized and upheld. Secondly, much effort should be directed toward developing fully compliant JDBC implementations (drivers). If these steps are achieved, the combination of Java and JDBC will provide a powerful, versatile tool in the development of distributed database interfaces.

Acknowledgements

We thank Yuri Breitbart for valuable suggestions. Research of the second author has been partially supported by US ARO grant DAAH 04-96-1-0398.

References

[1] Cornell, G. and Horstmann, C., Core Java, SunSoft Press, 1996.
[2] FastForward JDBC driver, Connect Software, Six months educational license.
[3] Flannagan, D. Java in a Nutshell, Reilly & Associates, 1996.
[4] Java Development Kit. http://www.javasoft.com.
[5] Jepson, B., Database Conncetivity: The Lure of Java, Java Report, 1997.
[6] JDBC Specifications, JavaSoft. http://splash.jacasoft.com/jdbc/index.html, 1996.
[7] Konopnicki, D., Shmueli, O., W3QS: A Query System for the WWW, VLDB-95. http://www.cs.technion.ac.il/ ˜konop/w3qs.html#publications
[8] Microline Component Toolkit, Microline Software. http://www.neurondata.com/index2.html, Free lite ed. license, 1997.
[9] MiniSQL, Hughes Techn. http://Hughes.com.au, 1997.
[10] Oaks S., Wong, H., Java Threads, O'Reilly, 1997. [11] Patel, P., and Moss, K., Java Database Programming with JDBC, Coriolis Group Book, 1996.
[12] Reese, G., mSQL-JDBC driver, Center for Imag. Environments, http://www.imaginary.com, 1997.
[13] Silberschatz, A., Korth, H.F. and Sudarshan, S., Database System Concepts, McGraw-Hill, 1997.
[14] Ullman, J. Database and Knowledge-Base Systems, Computer Science Press, 1988.
[15] Zloof, M.M., Query-by Example: a database language, IBM System Journal 16: 324–343, 1977.