A JDBC Driver Supporting Data Integration and Evolution

Jian Jia

University of Iowa, Iowa City, IA
jjia@cs.uiowa.edu

Introduction

The integration of information across different systems is one of the major challenges with current information systems. The fast growth of information technology has brought highly efficient ways of information management, but it also introduced many data sources with various structures, access restrictions and user applications. Migration of one data source into another is costly and time consuming, which makes it undesirable. This project involves automating database integration. Unity [1], a prototype system for relational database integration using semantic naming of database components, currently uses Open Data Base Connectivity (ODBC) to access distributed databases on different platforms. Java Data Base Connectivity (JDBC) implements API calls similar to ODBC using Java. JDBC is secure and can be easily downloaded from the network and installed on every platform. Almost every database vendor ships its software with its own JDBC driver. In order to extend and enhance the functionality of the Unity prototype to be widely applied in different environments, a Unity JDBC Driver with automatic integration capability will be developed in this project. The rest of this proposal presents motivation, a project summary, and then delves into details.

Motivation

Although extensive efforts have been performed on exploring information integration, integration of heterogeneous databases has not been resolved. Manually integrating information by direct querying of each database is commonly used at present. However, users must have knowledge of the schema of each distributed database source and completely understand all semantics and structures. In addition, database inconsistency is a major obstacle to generating an integrated view of queried results. Therefore, manually integrating information from different data sources does not truly resolve the problem. Automating the integration process provides a better solution, since users only see the integrated view of all data sources and are isolated from the complexity of information integration. The capability of capturing database semantics and their structure makes it possible to develop automated integration systems.

The integration system must be widely applicable and easily and quickly deployed across different platforms. Java, being object-oriented, robust, and portable, is an excellent language for database applications. The proposed project is to provide a universal integration system by implementing the integration functionality of Unity, a prototype integration system built using Visual C++ and runs on Windows OS, into a JDBC driver to enable its use with Java applications. Refer to Figure 1.

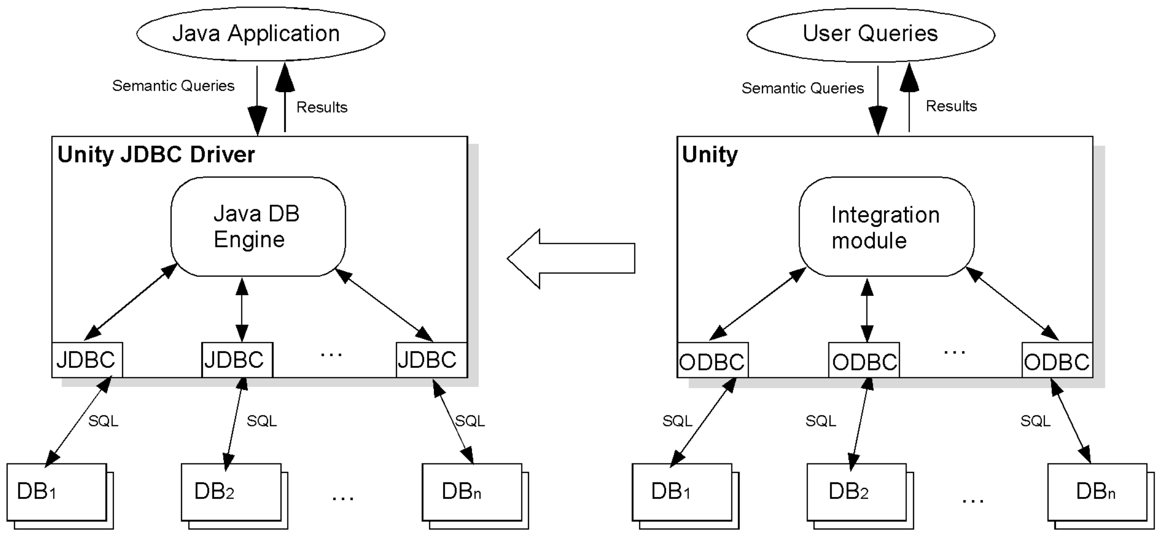

Figure 1. Migrating Unity From ODBC to JDBC

Goal

This project will produce a Unity JDBC Driver that is compliant with the JDBC 3.0 API and offers the ability to translate application requests into several database system specific SQL query requests. The driver will access each of the distributed database sources concurrently through the database specific JDBC Driver provided by the vendor. Query results from each database will be saved in memory and manipulated by a Database Engine. The integrated results will then be sent back to the application.

Project Details

Driver Types

JDBC Drivers can be categorized into four types:

1. JDBC-ODBC Bridges plus ODBC drivers: The Java Software bridge product provides JDBC access via ODBC drivers.
2. Native-API partly-Java drivers: These drivers convert JDBC calls to the native client API for Oracle, Sybase, Informix, IBM DB2, or other DBMSs.
3. JDBC-Net pure Java drivers: These drivers translate JDBC calls into a DBMS independent net protocol, which is then translated to a DBMS

protocol by a server. The net server middleware is able to connect its pure Java client to many different databases.

4. Native-protocol pure Java drivers: These drivers convert JDBC calls into the network protocol used by the DBMS.

The type 1 drivers rely on ODBC which reduces their portability and desirability for this project. Moreover, querying through the ODBC Bridge slows down transactions. The type 2 drivers are two-tier drivers that have to execute on a client's machine. The performance of the type 3 and the type 4 drivers are better than that of the first two types. The major advantage of the type 3 drivers is that the drivers do not need to be installed on the client machine. Further, they reduce the complexity as requests are passed through the network to the middle-tier server, which then translates the requests into the database-specific native-connectivity interface for the database server. The Unity JDBC driver is of type 4, since it access databases indirectly through their corresponding JDBC drivers, and does not use a middleware server to connect to the individual databases. However, the Unity JDBC driver will use the JDBC drivers for each underlying database which may be of type 3 or 4. Note that since the Unity JDBC driver does not use a middleware server, all integration of results is performed on the client machine.

Driver Structure and Implementation

Similar to other JDBC drivers, the Unity JDBC driver implements the core of a driver, using the interface *java.sql.Driver.* The *Driver* class is used to create a driver object. The JDBC driver should provide services such as establishing connections, sending queries to the data source, and processing the results. The following discussion introduces the major classes that construct the driver and enable its functions.

First, a set of classes: *java.sql.DatabaseMetaData, java.sql.DriverManager* and *java.sql.Connection,* is needed for opening a connection to the database. The *DriverManager* loads the appropriate driver. The *java.sql.Connection* class is used to create a connection object to access the database. The *DatabaseMetaData* returns information about the connection and information about the database to which the user has connected. The Unity JDBC driver should be able to create multiple connections to each distributed data source through its JDBC driver. Second, to execute a SQL query, *java.sql.Statement* and *java.sql.PreparedStatement* interfaces must be implemented. Finally, the class *java.sql.ResultSet* and *java.sql.ResultSetMetaData* should be used to obtain query results. In addition, driver security should be implemented. Any illegal connection to the database must be rejected.

In this project, a potential problem exists in query execution, as there are so many SQL variations that it is impossible for the Unity JDBC driver to support all of them. However, this will be resolved by having the driver support at least ANSI SQL-92 Entry Level. The extensions to SQL-92 Entry Level and SQL3 data types will be covered as further work. The current Unity JDBC driver is limited to data retrieving queries, as the functionality of automated updates on heterogeneous databases is still ongoing research, and thus is left as further work beyond this project.

What distinguishes the Unity JDBC driver from other JDBC drivers is its ability to translate user requests into data source specific SQL queries that map into different databases, and to integrate query results from multiple databases before they are sent back to user applications. A Query Translator, a Database Engine, and an Integration Module in the driver accomplish those functions. Implementation of those three modules is the most difficult and challenging part of this project, since it is not commonly performed in other JDBC drivers. The structure of the driver is presented in Figure 2.

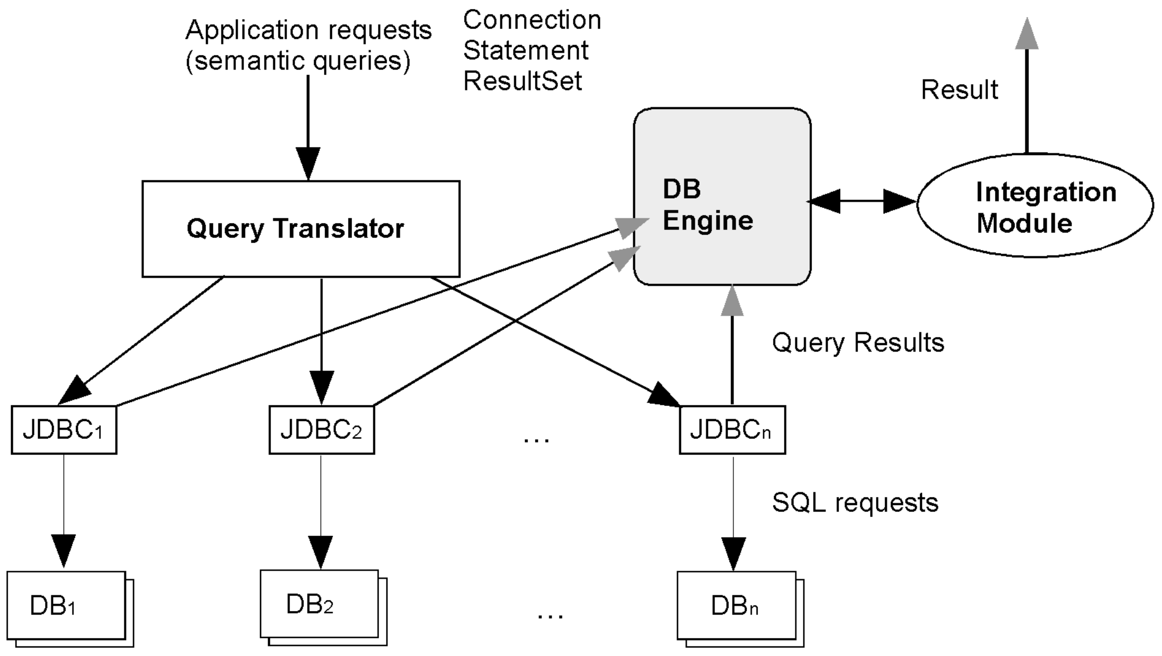

Figure 2. Inside the JDBC Driver

The Query Translator parses user query statements and translates them into SQL queries. This translation is accomplished using X-Spec documents saved in XML format describing each database. Unity has a similar translator written in C++. It will be used as the basis for developing the Java Query Translator. The Database Engine maintains retrieved results in memory. Initially, a free small Java main memory Data Base Engine (HSQLDB 1.7.0) is modified and embedded into the Unity JDBC driver. HSQLDB 1.7.0 is a small (about 100k), fast database engine, which offers both memory and disk based tables. It is written in pure Java, and can be downloaded from http://hsqldb.sourceforge.net. The future work will allow different database engines to be used. The Integration Module integrates the results based on a term matching algorithm. Integration is performed in Unity [2], by matching semantic names across databases. Semantic names are defined in the X-Spec for each database, and allow the integration system to match fields and tables representing identical concepts. Once this matching has been performed, the Integration Module performs data integration by matching keys and field values for all databases. This module is constructed by porting the Integration Module in Unity to Java.

Timeline

Phase I of this project involves building a pass-through JDBC driver that can be used to connect to a data source hosted by MySQL, through its JDBC driver, an open source type 4 driver. Query results are stored in the Unity JDBC driver. A test application program should successfully retrieve results from the Unity JDBC driver. This phase is estimated to take about one month.

In phase II, the Database Engine is embedded into the pass-through driver, so result sets can be stored appropriately in the Database Engine instead of in memory structures. This phase should be finished in two weeks.

The Query Translator is ported in phase III. User queries sent to the Unity driver are now expressed in a semantic query language instead of SQL. The Query Translator translates the semantic query into SQL, which is executed using the MySQL JDBC driver. Results are stored in the Database Engine, and returned to the user application from the Database Engine. Phase III will take about one month.

The Integration Module is ported in phase IV. This final phase involves handling multiple diverse databases and registering their drivers. The Query Translator converts semantic queries from the user to SQL queries for each database (as required). Results from each JDBC driver associated with each database are put in the Database Engine. The Integration Module then uses the Database Engine to combine results into one result set for the application. The final phase will take the reminder of the term. Finally, documentation and a project summary report will be produced.

Conclusion

In order to use the integration capabilities of Unity in a Java development environment, a Unity JDBC driver prototype will be developed in this project. This JDBC driver prototype offers the ability to access multiple database systems, and is capable of translating semantic user queries into SQL queries for each database system. Thus the complexities of accessing and integrating information from multiple databases are hidden from client applications. The project uses MySQL for testing. The flexibility of JDBC will allow the Unity JDBC driver to access other databases such as Sybase, Microsoft SQL server, Oracle and Informix on different platforms.

References

[1] R. Lawrence and K. Barker. Using Unity to Semi-Automatically Integrate Relational Schema, in *Proceedings of ICDE 2002*, Pages 329-330.

[2] R. Lawrence and K. Barker. Automated Integration of Relational Database Schemas, *University of Manitoba Technical Report TR-00-15.*