Uppsala Master's Theses in
Computer Science 291
2005-01-26

A JDBC driver for an Object – Oriented Database Mediator

Giedrius Povilavicius

Information Technology
Computer Science Department
Uppsala University
Box 337
S-751 05 Uppsala
Sweden

This work was carried out at
Uppsala Database Laboratory (UDBL)
Uppsala University
PO Box 513
SE-751 20 Uppsala
Sweden

Abstract

JDBC (JAVA Database Connectivity) is standard JAVA application programming interface (API) developed by JavaSoft. By using JDBC interface, JAVA applications and applets can access a wide variety of data sources using the same source code. Amos II (Active Mediator Object System) is a distributed mediator system that uses a functional data model and has a relationally complete functional query language, AMOSQL. The purpose of this work was to develop a JDBC driver for AMOS II. With help of it now it is possible to query AMOS II database from JAVA applications using standard interface, many simpler JDBC driven applications can be changed to use AMOS II just in couple of minutes.

Supervisor: prof. Tore Risch
Examiner: prof. Tore Risch

Passed:

Contents

1 Introduction

In recent years the need to effectively process large amounts of data has become increasingly important. Companies are using databases to store information about for example warehouses, orders, invoices, and much more. Because of this, the need to access this data using standard tools has become very important. Since there are many different kinds of databases in use today, a standard method for accessing data was developed for Windows Operating system by Microsoft, *Open Database Connectivity* (ODBC) [7]. Applications using ODBC as their data accessing method can read and manipulate databases for which there are ODBC drivers. Similar method was created and for JAVA programming language, *JAVA Database Connectivity* (JDBC) [6].

AMOS II (Active Mediator Object System) [8] is a light – weight, main – memory, object – relational database kernel, running on the Windows NT platform. It contains a relationally complete query language AMOSQL. The purpose of the AMOS project is to develop and demonstrate mediator architecture for supporting information systems where applications and users combine and analyze data from many different data sources. A data source can be a conventional database but also text files, data exchange files, web pages, programs that collect measurements or even programs that perform computations and other services.

Previously, the AMOS system lacked a nice interface to the user. The only way of manipulating the database was to write AMOSQL queries on the command-line. Later the more user friendly graphical interface was developed (Goovi). Next step in improving AMOS accessibility was development of easier programming interfaces such as ODBC-driver for AMOS. With help of it all ODBC applications can access the AMOS database in exactly the same way as they access other databases, for example Oracle or DB2. By creating JDBC driver for AMOS we will extend list of possible choices for developer with one more language - JAVA.

In this work JDBC driver for AMOS has been designed and implemented. As JDBC uses SQL language and AMOS uses AMOSQL certain translator has been used.

The goal of the project was to create as good implementation of JDBC driver as possible with current AMOS II system. It was not possible to create a fully JDBC compatible driver because of the limitations of SQL support in AMOS system and because of lack of certain features in AMOS JAVA interface. As will be shown later, the possibility to query AMOS system without SQL was also implemented and it allows user to manipulate data directly.

Chapters 2 and 3 provide background information about AMOS DBMS and JDBC. Chapters 4-5 describe driver design and implementation process. 6th, last chapter describes limitations of created driver. It also lists possible improvements and future work. Conclusions are also provided there.

As the main result of this project is JAVA source files (around 6000 lines of code) the appendix contains some more details about the implemented driver.

2 The AMOS DBMS

In this chapter more details about AMOS II system will be given.

2.1 AMOS II

Amos II is a distributed mediator system with an object oriented and functional data model. Queries to that data model are written in AMOSQL, a relationally complete functional query language. The system can consist of several autonomous and distributed Amos II peers. These peers can interoperate through its distributed multi-database facilities. Each mediator peer offers three possibilities to access data:

- Access to data stored in an Amos II database.
- Access to wrapped data sources.
- Access to data that is reconciled from other mediator peers.

Especially the second point makes Amos II extensible. New application oriented data types or operators can easily be wrapped. And it is possible to add them to the query language, AMOSQL. Thus a powerful query and data integration is offered by the Amos II system. First this chapter describes the mediator-wrapper approach, on which the Amos II system is based; later the Amos II architecture is described. As the main purpose of Amos II is using it as a database, the data model of Amos II is described. Finally as in this project work is done with JAVA programming language the two possible JAVA interfaces to access AMOS system are described.

2.2 The mediator-wrapper architecture

Today a lot of applications are developed that need to access databases. Therefore an increasing number of distributed databases are in use. Online travel agencies for example must be able to access databases of flight companies to gain information of empty seats, of airports to see arrival and departure timetables and of hotels to search for free rooms. Of course such information won’t be stored in one database, but each flight company, airport and hotel will have their own database. Thus access to several databases is necessary.

Furthermore there exists a large amount of possibilities to store data. One company having this database and that schema, the next company might have a completely different view on the stored data. Thus an application needs the possibility to access heterogeneous data sources.

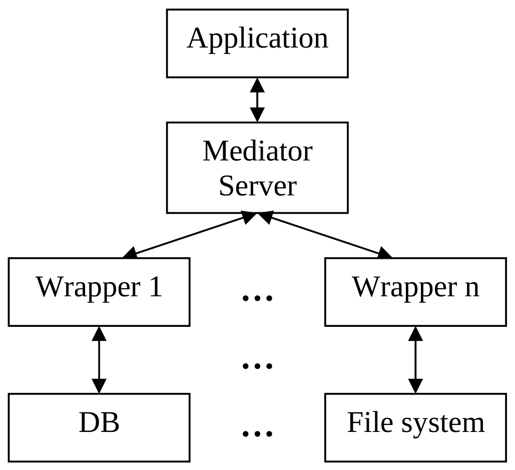

Figure 2.1: Mediator Wrapper Architecture

Therefore the mediator/wrapper approach gives a good support for applications to access distributed and heterogeneous data sources. But what is this approach?

A mediator system consists of a mediator server and one or several wrappers. The *mediator server* is a central software module that comprises a *common data model* (CDM). This CDM shares its domain-specific knowledge about data with higher levels of mediator layers or with applications. The task of a mediator is to answer queries that are sent from applications to the mediator's common data model. The query will be split depending on the data and capabilities of the target data sources. As a mediator can regard other mediators as data sources, a network of mediators, consisting of several layers, can be created. In such a network only *primitive* mediators should have access to data sources. Then higher layers, consisting of an advanced data abstraction, can be accessed by applications. One mediator can access different autonomous data sources, like databases, XML files or object stores, but never directly. Each data source can be accessed through software interfaces, called *wrapper* [10]. Queries, sent from the mediator, are translated by the wrapper to a data source specific format and thus hide the heterogeneity of that data source. After that the wrapper retrieves the query result, which has to be translated again into the common data model of the corresponding mediator. The translated data is passed to the mediator. There all results from all data sources are integrated and returned to the application.

2.3 The Amos II architecture

The Amos II system is based on the mediator/wrapper architecture. It is a distributed mediator system consisting of one or several mediator peers. These peers can communicate via the Internet using TCP/IP. Each peer offers its own virtual functional database layer, consisting of:

- Data abstractions that provide transparent functional views for clients and other mediator peers to access data sources,
- A functional query language: AMOSQL,
- A storage manager,
- A transaction manager.

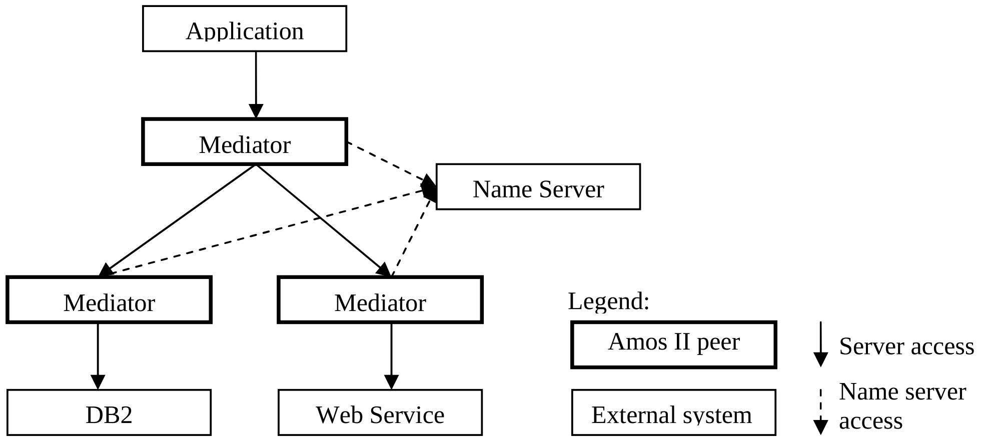

Figure 2.2: Example of an Amos II system with three mediators.

The core of Amos II is an open, light-weight and extensible database management system (DBMS). In Amos II it is possible to build up layers of peers with a dynamic communication topology. A *distributed mediator query optimizer* optimizes this communication topology. To compute an optimized execution plan for a given query, data and schema information are exchanged between the peers. In figure 2.2, the high level mediator defines mediating functional view integrating data from them. The views include facilities for semantic reconciliation of data retrieved from the two lower mediators. The two lower mediators translate data from a wrapped relational database and a web server, respectively. They have knowledge of how to translate AMOSQL queries to SQL through JDBC and, for the web server, to web service requests. To summarize this, figure 2.2 illustrates the distribution of mediator peers. Two distributed data sources offer through three distributed mediator peers their data to an application. Communication between peers is illustrated by thick lines, where the arrow indicates the peer running as server. One special mediator peer is listed in the figure, the *name server*. Every mediator peer must belong to a group and every group of mediator peers must have a name server. The name server stores meta-information like:

- Names of peers,
- Locations of peers,
- Additional data about all peers in that group.

It must be mentioned that the mediator peers forming a group are still autonomous and there exists no control schema on the name server. And it is left to each peer to describe its own local data view and data sources. In what way is the name server involved in peer communication? All communications are done through messages that can request or deliver data. To avoid a bottleneck the name server is only involved when a new mediator peer wants to register to the group. As soon as mediators are known to each other they can communicate directly without any information from the name server. In figure 2.2 the communication with the name server is illustrated by dotted lines. The name server always acts, with regard to the other mediator peers, as a server.

2.4 Data model

The basic components of the data model of Amos II are *objects*, *types* and *functions*. In object oriented programming languages these concepts approximately correspond to *classes*, *methods*, and *instances*. Types are used for classifying objects. Each object is an instance of a type. All properties of an object as well as relationships between objects are represented by functions

2.4.1 Types

For each entity type in an Entity-Relationship diagram an Amos II type is created. Types unite objects with similar properties. Furthermore multiple inheritance is offered, what means a supertype/subtype hierarchy can be built. An instance of a type is always an instance of all supertypes. If an object inherits from more than one type it gains all properties from all supertypes.

Two kinds of types exist, *stored* and *derived* types. Derived types are mainly used for reconciliation, while the stored types are defined and stored in an Amos II database. The following command creates a person and an assistant type in Amos II:

```
create type Person;
create type Assistant under Person;
```

Every Amos II peer offers a basic type hierarchy that can be seen in figure 2.3. The root element is named Object. All (system and user) defined type names are stored in a type named *Type*. Again all functions (described in chapter 2.4.2) are instances of a type named *Function*. When a user defines a type it is always a subtype of type *Userobject*.

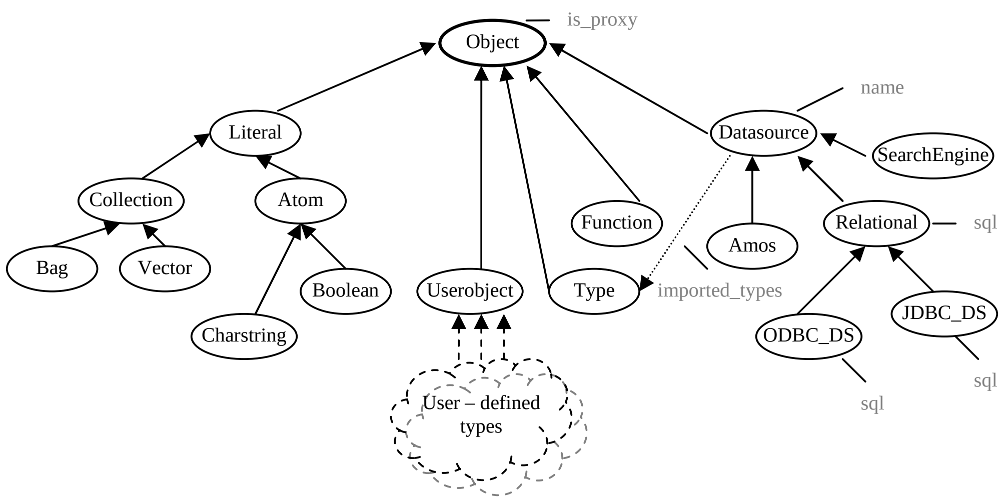

Figure 2.3: AMOS II system type hierarchy [8]

2.4.2 Functions

Functions model the relationship between objects, model properties of objects and computations over objects. In functional queries and views they are representing the basic primitives. As already mentioned, functions are instances of the type *Function*. The function's *signature* contains information about all arguments, such as the types and optional names, and about the result of the function. The next example shows the signature of a function modeling the attribute name of type *Person*:

name(Person)->Charstring

Furthermore, functions can be overloaded, i.e. functions defined for different combinations of arguments can have the same name (*polymorphism*). The selection of the correct function implementation of an overloaded function is made at function invocation based on the actual argument types.

2.4.3 Objects

Objects in Amos II are corresponding to Entities in an Entity-Relationship diagram and are instances of Amos II types. Everything in Amos II is represented as an object, independent whether the object is user-defined or system-defined. *Literals* and *surrogates* are the main kinds for representing objects. Literal objects are primitive objects like integers, strings or even *collections* that represent arrays of other objects. In addition to this are surrogates that are created by the user or the system and are describing real world entities. A *person*-object in Amos II can be created with a command as follows:

create person instances :thisisme;

When a query requests this object, the returned result will be displayed similar to this scheme:

#[OID 1101]

The system assigns unique object identifiers (OIDs) to all surrogate objects.

2.5 External AMOS II interfaces

Applications that are developed in programming languages and want to access the Amos II system requires special interfaces to the mediator layer. There are two ways to interface AMOS II with other programs, either an external program calls AMOS II through the *callin* interface, or AMOS II calls external functions through the *callout* interface [3]. Interfaces to several programming languages are already developed:

1. Lisp: Amos II can be called from a Lisp dialect, named *ALisp*. The AMOSQL parser is based on Lisp macros that translate queries and statements written in AMOSQL to Lisp code. The interface from ALisp to AMOS II is an *embedded query* interface. This method for accessing Amos II can be very fast and efficient.

2. C: The low level C interfaces are intended to be used by developers that need access to and extend the kernel of the system. This interface should be used for time critical applications.

3. JAVA: The most convenient way to write AMOS II applications is using the JAVA interface that is described in [4]. The JAVA interface is divided into a callin and callout interface. The use of these two interfaces is similar to the ones from C.

The JDBC driver is written in JAVA language and uses JAVA interface to communicate with AMOS DBMS.

As already mentioned there are two main kinds of external interfaces, the *callin* and the *callout* interfaces:

- With the *callin* interface a program in JAVA calls AMOS II. The callin interface is similar to the call level interfaces for relational databases, such as ODBC, JDBC, ORACLE call-level interface, etc.

- With the *callout* interface AMOS II functions call methods written in JAVA. Each such *foreign AMOS function* is implemented by one or several JAVA methods. The foreign functions in AMOS II are *multi-directional* which allow them to have separately defined inverses and to be indexed. The system furthermore allows the callin interface to be used also in foreign functions, which gives great flexibility and allows JAVA methods to be used as a form of *stored procedures*.

With the callin interface there are two ways to call AMOS II from JAVA:

- In the *embedded query* interface a JAVA method is provided that passes strings containing AMOSQL statements to AMOS II for dynamic evaluation. Methods are also provided to access the results of the dynamically evaluated AMOSQL statement. The embedded query interface is relatively slow since the AMOSQL statements have to be parsed and compiled at run time.

- In the *fast-path* interface predefined AMOS II functions are called as JAVA methods, without the overhead of dynamically parsing and executing AMOSQL statements. The fast-path is significantly faster than the embedded query interface. It is therefore recommended to always make AMOS II derived functions and stored procedures for the various AMOS II operations performed by the application and then use the fast-path interface to invoke them directly

There are also two choices of how the connection between application programs and AMOS II is handled:

1. AMOS II can be linked directly with a JAVA application program. This is called the *tight connection* where AMOS II is an *embedded database* in the application. It provides for the fastest possible interface between an application and AMOS II since both the application and AMOS II run in the same address space. The disadvantages with the tight connection are that AMOS II and the application must run on the same computer. Another disadvantage is that only a single JAVA application can be linked to AMOS II.

2. The JAVA application can be run as a client to an AMOS II server. This is called the *client-server connection*. With the client-server connection several applications can access the same AMOS II concurrently. The JAVA applications and the AMOS II server run as different programs. However, the overhead of calling AMOS II from another program can be several hundred times slower than the tight connection. Currently the client-server connection also requires AMOS II kernel code (in C) to be called from JAVA, and AMOS II JAVA clients are thus not applets. All the necessary code is provided through *JAVAamos.jar*.

Some advantages with the JAVA interfaces compared to the C and Lisp interfaces which should be mentioned also are:

1. Programming errors in the JAVA code cannot cause Amos II to crash. By contrast, programming errors in C can cause very nasty system errors. The Lisp interfaces are much safer than C, but not as protected (and limited) as the JAVA interfaces.

2. Memory is deallocated automatically in JAVA through its automatic garbage collector. In C/C++ memory deallocation is manual and very error prone, while Lisp also has automatic garbage collection.

3. JAVA has extensive libraries for user interfaces, web accesses, etc. which can easily be interfaced from AMOS II through its JAVA interfaces.

The disadvantages with the JAVA interfaces are that they are slower and more limited than the other ones.

3 JDBC

While many standards have been proposed to address the needs of multi-DBMS (Database Management System) access, ODBC has emerged as the de facto standard for the Microsoft Windows platform and is a component of Microsoft’s Windows Open Services Architecture (WOSA). Essentially all modern RDBMS (Relational Database Management System) products support this standard, either as their sole interface to the outside world or in addition to their own proprietary interface.

The Internet has spurred the invention of several new technologies in client/server computing – one of which is JAVA. JAVA is two-dimensional: it's a programming language and also a client/server system in which programs, are automatically downloaded and run on the local or client, machine. The wide acceptance of JAVA has prompted its quick development. JAVA includes JAVA compilers, interpreters, tools, libraries, and integrated development environments (IDE). JavaSoft is leading the way in the development of libraries to extend the functionality and usability of JAVA as a serious platform for creating applications. One of these libraries, called Application Programming Interfaces (APIs), is the JAVA Database Connectivity API, or JDBC. Its primary purpose is to allow developers to easily write programs which connect to databases in JAVA.

3.2 What is JDBC?

As already mentioned JDBC stands for JAVA Database Connectivity. It refers to several things, depending on its context:

- It's a specification for using data sources in JAVA applets and applications;
- It's an API for using low-level JDBC drivers.
- It's an API for creating the low-level JDBC drivers, which do the actual connecting/transacting with data sources.

Maybe it looks confusing but it's really quite simple: JDBC defines every aspect of making data-aware JAVA applications and applets. The low-level JDBC drivers perform the database-specific translation to the high-level JDBC interface. The developer uses this interface so he doesn't need to worry about the database-specific syntax when connecting to and querying different databases. The JDBC is a package, much like other JAVA packages such as e.g. JAVA.awt. It is currently a part of the standard JAVA Developer's Kit (JDK) distribution. It's also included in the standard part of the general JAVA API as the JAVA.sql package. The exciting aspect of JDBC is that the drivers necessary for connection to their respective databases do not require the client to pre-install anything: A JDBC driver can be downloaded along with an applet!

The JDBC project was started in January of 1996, and first specification was frozen in June of 1996. 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. Currently there is already 3rd version of JDBC API and, as we can see now from this list of vendors who have already endorsed the JDBC, it's sure to be widely accepted by the software industry (this is only a partial list): Borland International Inc., Informix Software Inc., Intersolv, Oracle Corporation, SAS Institute Inc., SCO, Sybase Inc., Symantec and Visigenic Software Inc.

The JDBC is heavily based on the ANSI SQL-92 standard, which specifies that a JDBC driver should be SQL-92 entry-level compliant to be considered a 100 percent JDBC-compliant driver (more about that in chapter 3.2.4). This is not to say that a JDBC driver has to be written for a SQL-92 database; a JDBC driver can be written for a legacy database system and still function perfectly. Even though some driver does not implement every single SQL-92 function, it is still a JDBC driver. This flexibility is a major selling point for developers who are bound to legacy database systems, but who still want to extend their client applications.

3.2.1 JDBC architecture

Different database systems have surprisingly little in common: just a similar purpose and a mostly compatible query language. Beyond that, every database has its own API that you must learn to write programs that interact with the database. This has meant that writing code capable of interfacing with databases from more than one vendor has been a daunting challenge. As mentioned before cross-database APIs exist, most notably Microsoft's ODBC API, but these tend to find themselves, at best, limited to a particular platform.

JDBC is Sun's attempt to create a platform-neutral interface between databases and JAVA. With JDBC, you can count on a standard set of database access features and (usually) a particular subset of SQL, SQL-92. The JDBC API defines a set of interfaces that encapsulate major database functionality, including running queries, processing results, and determining configuration information (see chapter 3.2.4). A database vendor or third-party developer writes a JDBC *driver*, which is a set of classes that implements these interfaces for a particular database system. An application can use a number of drivers interchangeably. Figure 3.1 shows how an application uses JDBC to interact with one or more databases without knowing about the underlying driver implementations. From there it’s obvious that JDBC can be divided into four parts:

- **Application** – performs some calculations and uses JDBC to submit SQL sentences and receive results.
- **Driver Manager** – responsible for correct driver initialization. It also provides tracing feature.
- **Driver -** submits SQL requests to a specific data source and returns results to the application.

- **DBMS** – provides data which user wants to access.

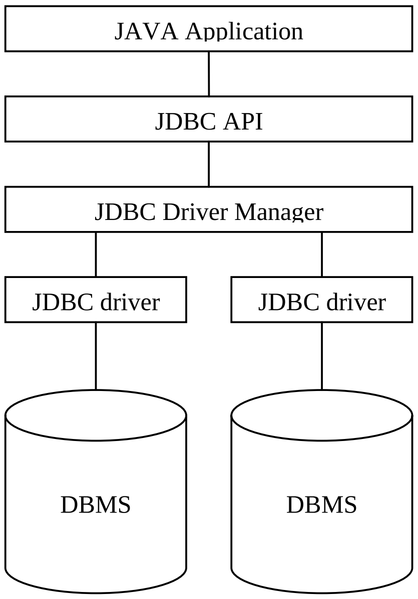

Figure 3.1 JDBC Structure

3.2.2 The Driver Manager class

The JDBC DriverManager is a static class that provides services to connect to JDBC drivers. The DriverManager is provided by JavaSoft and does not require the driver developer to perform any implementation. Its main purpose is to assist in loading and initializing a requested JDBC driver. Other than using the DriverManager to register a JDBC driver (registerDriver) to make itself known and to provide the logging facility, a driver does not interface with the DriverManager. In fact, once a JDBC driver is loaded, the DriverManager drops out of the picture altogether, and the application or applet interfaces with the driver directly.

3.2.3 Drivers

A driver is a set of classes that implements the necessary interfaces defined in JDBC API. Each driver is specific to a particular DBMS. The driver exposes features and capabilities of corresponding DBMS. If, for example, the DBMS does not supports outer joins, then neither should the driver. Of course certain features can be emulated by the driver (see chapter 5.3.2 for example).

Some tasks performed by the driver include:

- Connecting and disconnecting from DBMS;

- Submitting SQL statements for execution. The driver could also modify JDBC SQL into certain DBMS specific language if required (see chapter 5.2 for more details);
- Sending and receiving data to DBMS. This includes and necessary data conversions;
- Transaction control;
- Providing information about DBMS.
- Mapping DBMS specific errors to JDBC errors.

All drivers are divided into four categories in the way how they interact with Java and connect to DBMS (see chapter 4 for details). Besides that they can also be divided into two kinds, file-based drivers and DBMS-based drivers. File-based drivers access the database file directly (for example a driver for a simple text-file). In this case the driver acts as both driver and data source, it processes JDBC calls and SQL statements. This kind of driver has to implement its own database engine. DBMS-based drivers, on the other hand, access the physical data through a separate database engine. The driver only processes JDBC calls. SQL statements are passed to the database engine for processing, that is, the driver acts as the client in a client/server configuration where the DBMS acts as the server.

3.2.4 JDBC interfaces

The JDBC API specification provides a series of *interfaces* that must be implemented by the JDBC driver developer. An interface declaration creates a new reference type consisting of constants and abstract methods. An interface cannot contain any implementations—that is, executable code. What does all of this mean? The JDBC API specification dictates the methods and method interfaces for the API, and a driver must fully implement these interfaces. An application which uses JDBC makes method calls to the JDBC interface, not a specific driver. Because all JDBC drivers must implement the same interface, they are interchangeable.

There are a few rules that a developer must follow when implementing interfaces. First, he must implement the interface exactly as specified. This includes the name, return value, parameters, and throws clause. Secondly, he must be sure to implement all interfaces as public methods. He should remember that this is the interface that other classes will see; if it isn't public**,** it can't be seen. Finally, all methods in the interface must be implemented. If developer forgets some, the Java compiler kindly reminds him that.

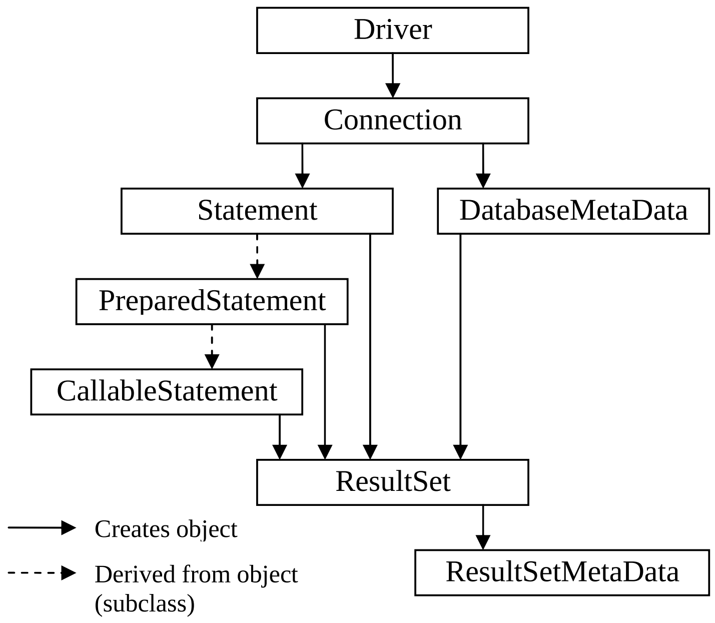

Figure 3.2 JDBC interfaces

Besides interfaces, a JDBC specification also provides certain set of classes which should be used when writing JDBC driver. DriverManager class was already mentioned in chapter 3.2.2 other useful classes are:

- DriverPropertyInfo – provides information about driver;
- Types – provides information about types supported by JDBC;
- Data, Time, Timestamp – classes required when dealing with certain data types;
- SQLException, SQLWarning, DataTruncation – special types of exceptions which should be used by the driver.

3.2.5 JDBC Compatible™

Naturally, there is no agreement on what is the proper set of functionality of a DBMS. Even databases that follow the relational model, uses SQL as its query language, and run on client-server architecture have no consensus on functionality. Different DBMS naturally have different functionality and users purchase the product partially due to the extended functionality that the product offers. Because of this usually it's impossible to implement full set of methods defined by the JDBC specification. To help people choose right JDBC driver SUN introduced *JDBC Compatible* trademark. Any developer of certain driver can test his work with help of *JDBC API test suite* which is freely available on internet. If driver passes all tests it will be putted on a special website (http://servlet.java.sun.com/products/jdbc/drivers) and any JDBC developer will be able to find it there.

3.3. Basic flow of a JDBC application

This part describes how a JDBC-based application connects to a data source, submits queries, and fetches results.

First of all the application (or user) chooses to which driver it wishes to connect. That is done with help of simple URL (*Uniform Resource Locator*): *jdbc:subprotocol:subname*. Where subprotocol is particular database connectivity mechanism supported and subname is defined by JDBC driver. For example, the format for MYSQL DBMS driver could be:

jdbc:mysql://www.somewebsite.com:4333/db_web

The DriverManager tries all drivers known to it, finds out which of them supports this URL (there can be and more then one) and returns Connection object. On initialization of this object connection to database is established.

With help of connection object all the other resources can be received. To send some query to the database Statement, PreparedStatement, or CallableStatement objects could be used. The main difference between those three objects is how they are executed by the DBMS.

Statement object which implements direct execution is the simplest option. It just sends a query to the DBMS for execution, for example a simple select statement. No parameters are allowed, that is, all necessary data has to be sent at once. This type of execution might be appropriate for simple queries which are expected to be executed only once. Since the DBMS has to compile the query before execution, this method is rather slow, at least if the same query is asked many times. In this case, prepared execution is significantly faster.

PreparedStatement and CallableStatement objects implements prepared queries support. First query is sent to the data source for preparation (compilation). Then query can be executed at a later time. It is also possible to send parameters to the DBMS before execution. An example: The select statement

SELECT age FROM person WHERE name = ?

where '?' is a parameter marker might be sent to the DBMS. Before execution, the value of the parameter is sent. After retrieving the result of the query new parameters can be sent for the next execution of the query. The query never has to be compiled again, at least not until its associated statement object has been closed.

Retrieving of results is done in a loop (fetch the next tuple if there is one) and at the same time type conversion is made if necessary.

When the application wishes to disconnect from the DBMS all statement and result set objects are closed (a connection can have many concurrent statements and many result sets), then connection object is closed. At this point the application is completely disconnected from the data source.

4 Driver design

The JDBC standard specifies what a JDBC driver should do and what interfaces it must implement. However, it does not specify how this should be done. In this chapter we will look at four different classic JDBC driver types and finally we will describe the chosen one.

4.1 JDBC driver types

All possible JDBC drivers fit into one of the four different categories (see Fig. 4.1). They vary in how they interact with JAVA and in their network capability. Each type has pros and cons.

4.1.1 Type I

This driver is also known as JDBC – ODBC bridge. It is included in the standard JDK 1.1 distribution and can be accessible as the sun.jdbc.odbc.JdbcOdbc Driver. This driver uses native methods and requires ODBC installed on the machine accessing this driver. Because of the used native methods this driver has certain limitations. Main one is that it cannot be used from applets.

The JDBC-ODBC bridge also requires different ODBC drivers for different databases installed. Overall, this type of driver is not flexible and can’t be deployed over the network.

4.1.2 Type II

This driver also uses native method calls to a database’s client library to make the data connection. This could be a very good interim solution, because client libraries for databases generally come with the database software. Vendors write JDBC driver that translates from the JDBC API to the specific client library. This type of driver also has certain restrictions. It uses native methods so the driver and the native executables must be pre-installed on the client machines. On the other hand this driver is also quiet fast because he is using native methods.

4.1.3 Type III

This driver is 100% pure JAVA, with no native method bindings, and it can be deployed with JAVA application/applet with no pre-installation. This driver uses some protocol for connection to a middleware server. The middleware server then takes the JDBC API calls from the JAVA client and translates them to DBMS specific calls.

This type of driver has good portability and requires no pre installation on client machine. Besides middleware server could provide special features such as security, caching or connection pooling. Of course disadvantage of such extra layer would be slow speed.

4.1.4 Type IV

This driver allows some JAVA application/applet to connect directly to a database server. It does not use native methods and is 100 percent pure JAVA. The JDBC driver translates calls to the specific database protocol and allows direct querying of the database.

Same as type III driver this driver is very portable and requires no pre installation. At the same time it is the fastest driver.

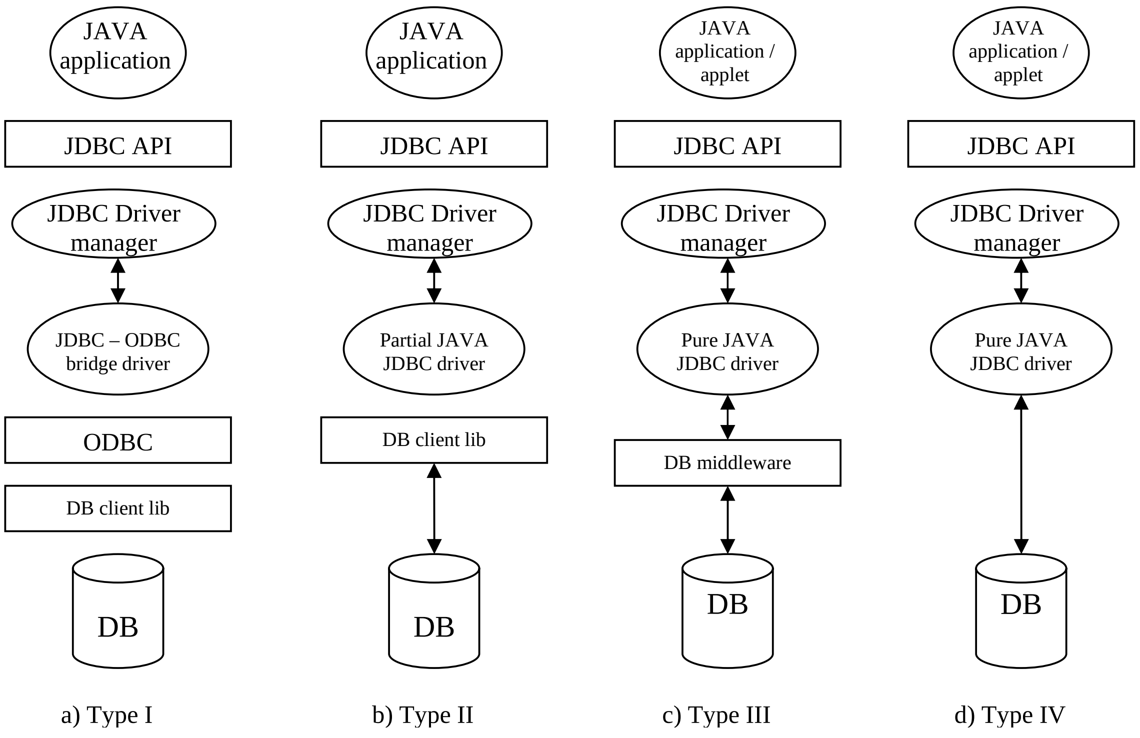

Figure 4.1: Four classical types of JDBC driver architecture

4.2 The chosen architecture

It’s clear that the best driver architecture choice could be driver of type III or IV, but at the same time they are also and more difficult to implement. As one of the biggest problems in their development it could be mentioned communication over the Internet with middleware (type III driver) or directly with DBMS (type IV driver). The AMOS II system lacks a pure JAVA communication protocol, which could be used in this case. Thats why it was decided to use AMOS JAVA *callin* interface (see chapter 2.5). With it we can implement two types of connection to database (see Figure 4.2).

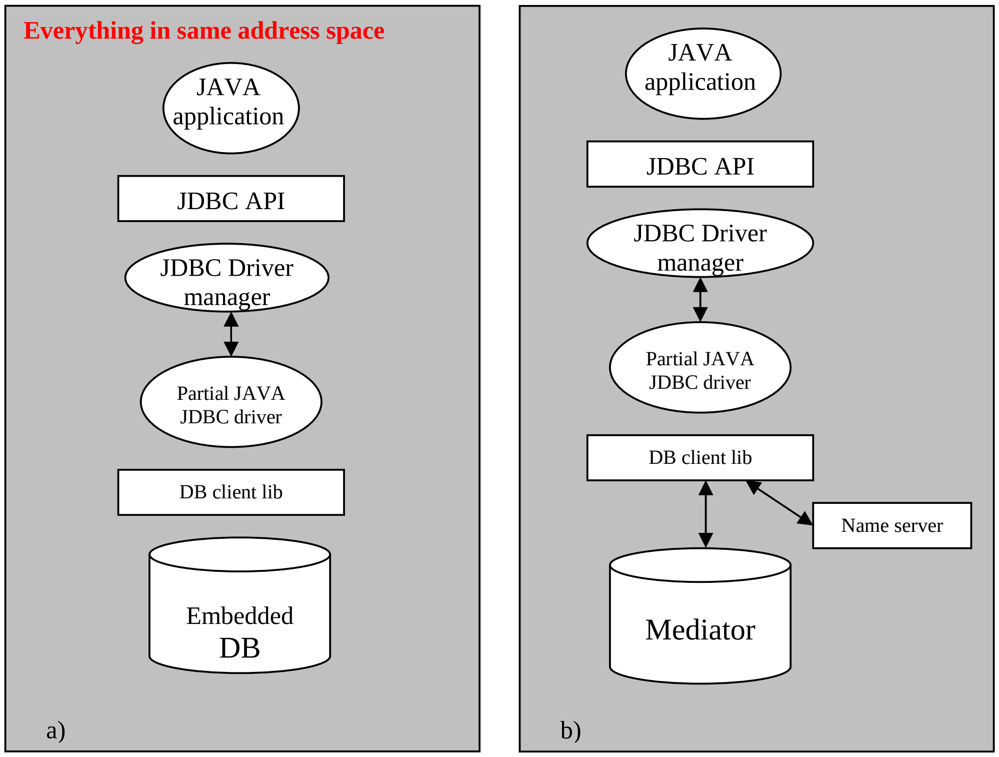

Figure 4.2: Chosen driver architecture: a) Connected to embedded database b) Connected to mediator peer with help of name server

If we look carefully into this solution we will see that this is modified type II driver. In first case driver’s connection to AMOS II is ultra-fast, since the system runs in the same address space as the application and because AMOS II is a main memory system. The downside can be the fact that embedded AMOS II is a single user system. The application would have its own local, private database. Such system could be very useful for application testing purposes.

In second case we would be able to connect to databases running on other machines, but communication speed would be much slower. Such system configuration is closer to real life situations.

As will be shown later both possibilities were provided in our implementation.

5 Implementation

Implementing a JDBC driver consists mostly of writing JAVA-code, which is not very interesting to read about. Instead, this chapter covers parts of the implementation process which differs somewhat from the development of an "ordinary" relational database driver.

5.1 Connecting to AMOS

Connection to Amos II DBMS is done with help of *callin* interface. As mentioned in chapter 2 there are two types of connections: *tight connection* and *client - server connection.* Both were made accessible from driver. DBMS to which we want to connect is chosen from URL. Extra data is provided with help of properties:

```
Properties info = new Properties();
info.setProperty("dbName","server");
info.setProperty("nameServerHost","127.0.0.1");

info.setProperty("dumpfile","amos2.dmp");
info.setProperty("directory","c:\\amosnt\\bin\\");

conn = DriverManager.getConnection("jdbc:amos:///amos2",info);
```

First two properties are required for connection to some AMOS DBMS. If *dbName* property is set to empty string then it represents a tight connection to the embedded database. Otherwise *dbName* must be the name of an AMOS II mediator peer known to the AMOS II nameserver. Ip address of name server is set by second property: *nameServerHost*.

Next two properties are required for JAVA *callin* interface. With their help developer provides information were local database dump file is located.

5.2 Translating SQL to AMOSQL

As mentioned before JDBC requires use of SQL language for querying database. If DBMS is using some other language then driver has to translate SQL sentences into DBMS native ones. Other possible solution for such problem is adding SQL front-end on DBMS side. AMOS II system for which this driver was implemented is using AMOSQL. Because primary task of this project isn't translation the work done by other students was used.

First versions of this driver used SQL Parser by Marcus Eriksson [7]. This parser works on driver side and is quite limited:

- Only select sentences are supported;
- No internal information from DBMS is used;
- It's written in C so native calls are used.

That's why later SQL Front by Markus Jägerskogh was tried. The main difference of this solution from SQL Parser is that all the work is done on a AMOS II server. Besides that much bigger part of SQL standard is supported.

Only implementation of driver which uses SQL Front will be discussed below.

5.3 Executing SQL statements

This chapter describes how the driver implements query execution and retrieving of results from executed statements.

5.3.1 Direct execution

Direct execution is simplest execution method. Driver does no preparations or modifications to SQL sentences. Here is the code snippet:

```
func = callinConnection.getFunction("charstring.sql->vector");
Tuple argl =  new Tuple();
argl.setArity(1);
argl.setElem(0,sql);
scan=callinConnection.callFunction(func,argl);
<snip>
res = new AmosJdbcResultSet(scan,AmosJdbcDefine.sentenceSQLFront);
```

Because SQL Front is implemented as an AMOS II foreign function it is accessed with help of fast-path interface. First we find AMOS II function which implements SQL Front and store reference to it in variable *func*. Next we set required parameters for that function. After executing SQL Front we get back list of results in variable *scan*. From it we can create ResultSet object.

5.3.2 Preparing queries for execution

By definition prepared queries are the fastest way to execute the same query many times, even with different variables for every execution. The query is prepared (compiled) on the data source and executed at a later time. The query remains active until the corresponding statement object is closed.

The problem with such queries in this driver development was that SQL Front has no support for prepared queries. There were two possible solutions for that:

- To leave this interface not implemented;
- Emulate its work on a driver.

First solution of course is the easiest, but problem is that many developers like to use prepared queries in their applications, because usually they are much faster then a direct execution. Because of that second solution was chosen and implemented.

On preparation of a query no data is send to the database. All what is done is analysis of the query with help of which we figure out how many parameters query has, e.g.

*SELECT * FROM person WHERE (age >?) and (name=?)*

has 2 parameters which should be set before execution. Hash table of size 2 is initialized for further use.

5.3.3. Binding parameters

Before executing a prepared query argument values, if any, must be supplied. PreparedStatement interface defines certain set of methods to do that:

- setInt
- setString
- setFloat
- setBoolean
- etc.

On use of any of these methods provided value is translated into string and saved in a hash table. Finally when all of them are set prepared statement can be executed. It is done with help of direct execution explained before. But first SQL query should be translated into sentence without parameters. It is done with help of this code:

```
for (int i=0;i<sql.length();i++)
    {
        if (sql.charAt(i)=='?')
        {
            preparedSql=preparedSql+(String)Parameters.get(new Integer(counter));
            counter++;
        }
        else
        {
            preparedSql=preparedSql+sql.charAt(i);
        }
```

}

5.4 Data coercion

At the heart of every JDBC driver is data as the whole purpose of the driver is to provide data. Actually, not only providing it but providing it in requested format. This is what data coercion is all about – converting data from one format to another. JDBC specifies the list of necessary conversions (see Figure 5.1)

In order to provide reliable data coercion, a data wrapper class was created (*AmosJdbcCommonValue*). This class contains a data value in some known format and provides methods to convert it to some specific format. *Object* class is used for this task:

Object data = (Object) s;
int internalType = Types.VARCHAR;

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
TINYINT	■	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓	▓						
SMALLINT	▓	■	░	░	░	░	░	░	░	░	░	░	░						
INTEGER	▓	░	■	░	░	░	░	░	░	░	░	░	░						
BIGINT	▓	░	░	■	░	░	░	░	░	░	░	░	░						
REAL	▓	░	░	░	■	░	░	░	░	░	░	░	░						
FLOAT	▓	░	░	░	░	■	░	░	░	░	░	░	░						
DOUBLE	▓	░	░	░	░	░	■	░	░	░	░	░	░						
DECIMAL	▓	░	░	░	░	░	░	■	░	░	░	░	░						
NUMERIC	▓	░	░	░	░	░	░	░	■	░	░	░	░						
BIT	▓	░	░	░	░	░	░	░	░	■	░	░	░						
CHAR	▓	░	░	░	░	░	░	░	░	░	■	░	░	▓	▓	▓	▓	▓	▓
VARCHAR	▓	░	░	░	░	░	░	░	░	░	░	■	░	▓	▓	▓	▓	▓	▓
LONGVARCHAR	▓	░	░	░	░	░	░	░	░	░	░	░	■	▓	▓	▓	▓	▓	▓
BINARY											▓	▓	▓	■	▓	▓			
VARBINARY											▓	▓	▓	▓	■	▓			
LONGVARBINARY											▓	▓	▓	▓	▓	■			
DATE											▓	▓	▓				■		▓
TIME											▓	▓	▓					■	▓
TIMESTAMP											▓	▓	▓				▓	▓	■

Figure 5.1 JDBC specified list of required conversions. Light grey color shows implemented transformations in this driver.

5.5 Accessing AMOS directly

Because of certain limitations in SQL Front it was decided to also provide the possibility to access AMOS DBMS directly. It is done simple by adding “amosql:” in front of sentence we want to execute.

6. Conclusion and future work

This chapter discusses the result of the work as well as limitations and possible improvements that can be made in the future.

6.1 Conclusion

In this project a JDBC driver for AMOS II was implemented. With help of it now it is possible to establish connections either to one embedded database via a tight connection or to one or several databases where the Amos II system runs as a server using standard JDBC API. This driver allows all usual JDBC features:

- Connection to database;
- Executing SQL sentences;
- Receiving results;
- Getting information about connected database.

6.2 Limitations

Driver implementation still has lots of limitations. Most of them are quite small and could be easily avoided by developer who writes programs carefully.

Possible improvements and future work are provided in chapter 6.3.

6.3 Future work

There are a lot of things that could be improved. Not only that this driver would be fully JDBC compliant, but also and driver architecture could be revisited and improved so that JAVA applets could be created also. The most important possible changes in my opinion are:

- To modify this driver so that it would be type 3 or 4 driver.
- Better SQL support is required to pass JDBC compliance tests provided by Sun.
- Java interface should be extended so that all the required data for JDBC driver could be acquired, e.g. getting column names for result sets.

All three tasks are quite big and time consuming. The first one is not easy to accomplish. To create driver of 4th type JAVA stub interface to Amos II client-server communication protocol will be required. To create type 3 driver certain communication protocol will be needed, good side is that code from this project driver implementation could be used as a starting point for middleware server so that no work with AMOS JAVA interface will be required. A second task will be difficult also because mapping from relational tables to objects is quite problematic. The last task is simpler but because writing native calls is error prone its implementation could take lots of time too.

References:

[1] Staffan Flodin, Martin Hansson, Vanja Josifovski, Timour Katchaounov, Tore Risch, and Martin Sk¨old: Amos II Release 6 User's Manual. *UDBL Technical Report, Dept. of Information Science, Uppsala University, Sweden*, March 27, 2004, http://user.it.uu.se/~udbl/amos/doc/amos_users_guide.html .

[2] Tore Risch: AMOS II Functional Mediators for Information Modelling, Querying, and Integration. *UDBL whitepaper*, *Dept. of Information Science*, *UppsalaUniversity*, *Sweden*. http://user.it.uu.se/~udbl/amos/amoswhite.html .

[3] Tore Risch: Amos II External Interfaces. *UDBL Technical Report, Dept. of Information Science, Uppsala University, Sweden*, 2001, http://user.it.uu.se/~torer/publ/external.pdf .

[4] Daniel Elin, Tore Risch: Amos II JAVA Interfaces. *UDBL Technical Report, Dept. of Information Science, Uppsala University, Sweden*, 2000, http://user.it.uu.se/~torer/publ/JAVAapi.pdf .

[5] Brian Jepson: JAVA database programming. *Wiley computer Publishing,* ISBN 0-471-16518-2, 1997

[6] Pratik Patel, Karl Moss: JAVA database programming with JDBC, 2nd edition. ISBN 1-57610-159-2, 1997

[7] Marcus Eriksson: An ODBC-driver for the mediator database AMOS II *Linköping Studies in Science and Technology, Master's Thesis No: LiTH-IDA-Ex-99/50*, 1999 http://user.it.uu.se/~udbl/Theses/MarcusErikssonMSc.pdf

[8] T.Risch, V.Josifovski, and T.Katchaounov: Functional Data Integration in a Distributed Mediator System, in P.Gray, L.Kerschberg, P.King, and A.Poulovassilis (eds.): Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data, *Springer*, ISBN 3-540-00375-4, 2003, http://user.it.uu.se/~torer/publ/FuncMedPaper.pdf

[9] M.Nyström and T.Risch: Engineering Information Integration using Object-Oriented Mediator Technology, *Software - Practice and Experience J., Vol. 34, No. 10, pp 949-975, John Wiley & Sons, Ltd.*, 2004.

[10] H.Lin, T.Risch, and T.Katchanounov: Adaptive data mediation over XML data. In special issue on Web Information Systems Applications of Journal of Applied System Studies (JASS), *Cambridge International Science Publishing, 3(2)*, 2002, http://user.it.uu.se/~torer/publ/jass01.pdf

[11] T. Katchaounov, T. Risch, and S. Zürcher: Object-Oriented Mediator Queries to Internet Search Engines, *International Workshop on Efficient Web-based Information Systems (EWIS), Montpellier, France, September 2nd*, 2002.

http://www.csd.uu.se/~torer/publ/orwise-EWIS-2002-final.pdf

[12] M.Koparanova and T.Risch: Completing CAD Data Queries for Visualization, International *Database Engineering and Applications Symposium (IDEAS 2002), Edmonton, Alberta, Canada*, July 17-19, 2002, http://www.csd.uu.se/~torer/publ/EMQ_f1.pdf

[13] G. Fahl and T. Risch: Query Processing over Object Views of Relational Data, *The VLDB Journal , Vol. 6 No. 4*, November 1997, pp 261-281 http://user.it.uu.se/~udbl/publ/vldbj97.pdf

APPENDIX A: Implemented JDBC classes and their methods

In this appendix all implemented JDBC classes and their methods are presented. More details about them can be found in JAVA SDK documentation.

Driver

acceptsURL

Returns true if the driver thinks that it can open a connection to the given URL.

connect

Attempts to make a database connection to the given URL.

getMajorVersion

Gets the driver's major version number.

getMinorVersio

Gets the driver's minor version number.

getPropertyInfo

Gets information about the possible properties for this driver.

jdbcComplian

Reports whether this driver is a JDBC COMPLIANT^{TM} driver.

Connection

close

Releases a Connection's database and JDBC resources immediately instead of waiting for them to be automatically released.

commit

Makes all changes made since the previous commit/rollback permanent and releases any database locks currently held by the Connection.

createStatement

Creates a Statement object for sending SQL statements to the database.

createStatement

Creates a Statement object that will generate ResultSet objects with the given type and concurrency.

getAutoCommit

Gets the current auto-commit state.

getMetaData

Gets the metadata regarding this connection's database.

getTransactionIsolation

Gets this Connection's current transaction isolation level.

isClosed

Tests to see if a Connection is closed.

isReadOnly

Tests to see if the connection is in read-only mode.

nativeSQL

Converts the given SQL statement into the system's native SQL grammar.

prepareStatement

Creates a PreparedStatement object for sending parameterized SQL statements to the database.

rollback

Drops all changes made since the previous commit/rollback and releases any database locks currently held by this Connection.

setAutoCommit

Sets this connection's auto-commit mode.

setTransactionIsolation

Attempts to change the transaction isolation level to the one given.

Statement

close

Releases this Statement object's database and JDBC resources immediately instead of waiting for this to happen when it is automatically closed.

executeQuery

Executes an SQL statement that returns a single ResultSet object.

executeUpdate

Executes an SQL INSERT, UPDATE or DELETE statement.

getConnection

Returns the Connection object that produced this Statement object.

getMaxRows

Retrieves the maximum number of rows that a ResultSet object can contain.

getResultSet

Returns the current result as a ResultSet object.

setMaxRows

Sets the limit for the maximum number of rows that any ResultSet object can contain to the given number.

PreparedStatement

clearParameters

Clears the current parameter values immediately.

executeQuery

Executes the SQL query in this PreparedStatement object and returns the result set generated by the query.

executeUpdate

Executes the SQL INSERT, UPDATE or DELETE statement in this PreparedStatement object.

getMetaData

Gets the number, types and properties of a ResultSet object's columns.

setBigDecimal

Sets the designated parameter to a java.math.BigDecimal value.

setBoolean

Sets the designated parameter to a Java boolean value.

setDouble

Sets the designated parameter to a Java double value.

setFloat

Sets the designated parameter to a Java float value.

setInt

Sets the designated parameter to a Java int value.

setLong

Sets the designated parameter to a Java long value.

setShort

Sets the designated parameter to a Java short value.

setString

Sets the designated parameter to a Java String value.

ResultSet

close

Releases this ResultSet object's database and JDBC resources immediately instead of waiting for this to happen when it is automatically closed.

findColumn

Maps the given ResultSet column name to its ResultSet column index.

getBigDecimal

Gets the value of the designated column in the current row of this ResultSet object as a java.math.BigDecimal with full precision.

getBoolean

Gets the value of the designated column in the current row of this ResultSet object as a boolean in the Java programming language.

getDouble

Gets the value of the designated column in the current row of this ResultSet object as a double in the Java programming language.

getFloat

Gets the value of the designated column in the current row of this ResultSet object as a float in the Java programming language.

getInt

Gets the value of the designated column in the current row of this ResultSet object as an int in the Java programming language.

getLong

Gets the value of the designated column in the current row of this ResultSet object as a long in the Java programming language.

getMetaData

Retrieves the number, types and properties of this ResultSet object's columns.

getObject

Gets the value of the designated column in the current row of this ResultSet object as an Object in the Java programming language.

getShort

Gets the value of the designated column in the current row of this ResultSet object as a short in the Java programming language.

getString

Gets the value of the designated column in the current row of this ResultSet object as a String in the Java programming language.

next

Moves the cursor down one row from its current position.

ResultSetMetaData

getColumnCount

Returns the number of columns in this ResultSet object.

getColumnLabel

Gets the designated column's suggested title for use in printouts and displays.

getColumnName

Get the designated column's name.

getColumnType

Retrieves the designated column's SQL type.

getColumnTypeName

Retrieves the designated column's database-specific type name.

getPrecision

Get the designated column's number of decimal digits.

getScale

Gets the designated column's number of digits to right of the decimal point.

getSchemaName

Get the designated column's table's schema.

getTableName

Gets the designated column's table name.

isAutoIncrement

Indicates whether the designated column is automatically numbered, thus read-only.

isCaseSensitive

Indicates whether a column's case matters.

isCurrency

Indicates whether the designated column is a cash value.

isDefinitelyWritable

Indicates whether a write on the designated column will definitely succeed.

isNullable

Indicates the nullability of values in the designated column.

isReadOnly

Indicates whether the designated column is definitely not writable.

isSearchable

Indicates whether the designated column can be used in a where clause.

isSigned

Indicates whether values in the designated column are signed numbers.

isWritable

Indicates whether it is possible for a write on the designated column to succeed.

DatabaseMetaData

allTablesAreSelectable

Can all the tables returned by getTable be SELECTed by the current user?

dataDefinitionCausesTransactionCommit

Does a data definition statement within a transaction force the transaction to commit?

dataDefinitionIgnoredInTransactions

Is a data definition statement within a transaction ignored?

doesMaxRowSizeIncludeBlobs

Did getMaxRowSize() include LONGVARCHAR and LONGVARBINARY blobs?

getColumns

Gets a description of table columns available in the specified catalog.

getDatabaseProductName

What's the name of this database product?

getDatabaseProductVersion

What's the version of this database product?

getDefaultTransactionIsolation

What's the database's default transaction isolation level? The values are defined in java.sql.Connection.

getDriverMajorVersion

What's this JDBC driver's major version number?

getDriverMinorVersion

What's this JDBC driver's minor version number?

getDriverName

What's the name of this JDBC driver?

getDriverVersion

What's the version of this JDBC driver?

getExtraNameCharacters

Gets all the "extra" characters that can be used in unquoted identifier names (those beyond a-z, A-Z, 0-9 and _).

getIdentifierQuoteString

What's the string used to quote SQL identifiers? This returns a space " " if identifier quoting isn't supported.

getMaxCatalogNameLength

What's the maximum length of a catalog name?

getMaxCharLiteralLength

What's the max length for a character literal?

getMaxColumnNameLength

What's the limit on column name length?

getMaxColumnsInGroupBy

What's the maximum number of columns in a "GROUP BY" clause?

getMaxColumnsInOrderBy

What's the maximum number of columns in an "ORDER BY" clause?

getMaxColumnsInSelect

What's the maximum number of columns in a "SELECT" list?

getMaxColumnsInTable

What's the maximum number of columns in a table?

getMaxConnections

How many active connections can we have at a time to this database?

getMaxRowSize

What's the maximum length of a single row?

getMaxSchemaNameLength

What's the maximum length allowed for a schema name?

getMaxStatementLength

What's the maximum length of an SQL statement?

getMaxStatements

How many active statements can we have open at one time to this database?

getMaxTableNameLength

What's the maximum length of a table name?

getMaxTablesInSelect

What's the maximum number of tables in a SELECT statement?

getMaxUserNameLength

What's the maximum length of a user name?

getNumericFunctions

Gets a comma-separated list of math functions.

getPrimaryKeys

Gets a description of a table's primary key columns.

getSchemas

Gets the schema names available in this database.

getSchemaTerm

What's the database vendor's preferred term for "schema"?

getSQLKeywords

Gets a comma-separated list of all a database's SQL keywords that are NOT also SQL92 keywords.

getStringFunctions

Gets a comma-separated list of string functions.

getSystemFunctions

Gets a comma-separated list of system functions.

getTables

Gets a description of tables available in a catalog.

getTableTypes

Gets the table types available in this database.

getTypeInfo

Gets a description of all the standard SQL types supported by this database.

getURL

What's the url for this database?

getUserName

What's our user name as known to the database?

isCatalogAtStart

Does a catalog appear at the start of a qualified table name? (Otherwise it appears at the end)

isReadOnly

Is the database in read-only mode?

nullPlusNonNullIsNull

Are concatenations between NULL and non-NULL values NULL? For SQL-92 compliance, a JDBC technology-enabled driver will return true.

nullsAreSortedAtEnd

Are NULL values sorted at the end regardless of sort order?

nullsAreSortedAtStart

Are NULL values sorted at the start regardless of sort order?

nullsAreSortedHigh

Are NULL values sorted high?

nullsAreSortedLow

Are NULL values sorted low?

storesLowerCaseIdentifiers

Does the database treat mixed case unquoted SQL identifiers as case insensitive and store them in lower case?

storesLowerCaseQuotedIdentifiers

Does the database treat mixed case quoted SQL identifiers as case insensitive and store them in lower case?

storesMixedCaseIdentifiers

Does the database treat mixed case unquoted SQL identifiers as case insensitive and store them in mixed case?

storesMixedCaseQuotedIdentifiers

Does the database treat mixed case quoted SQL identifiers as case insensitive and store them in mixed case?

storesUpperCaseIdentifiers

Does the database treat mixed case unquoted SQL identifiers as case insensitive and store them in upper case?

storesUpperCaseQuotedIdentifiers

Does the database treat mixed case quoted SQL identifiers as case insensitive and store them in upper case?

supportsAlterTableWithAddColumn

Is "ALTER TABLE" with add column supported?

supportsAlterTableWithDropColumn

Is "ALTER TABLE" with drop column supported?

supportsANSI92EntryLevelSQL

Is the ANSI92 entry level SQL grammar supported? All JDBC CompliantTM drivers must return true.

supportsANSI92FullSQL

Is the ANSI92 full SQL grammar supported?

supportsANSI92IntermediateSQL

Is the ANSI92 intermediate SQL grammar supported?

supportsCatalogsInDataManipulation

Can a catalog name be used in a data manipulation statement?

supportsCatalogsInIndexDefinitions

Can a catalog name be used in an index definition statement?

supportsCatalogsInPrivilegeDefinitions

Can a catalog name be used in a privilege definition statement?

supportsCatalogsInProcedureCalls

Can a catalog name be used in a procedure call statement?

supportsCatalogsInTableDefinitions
Can a catalog name be used in a table definition statement?

supportsColumnAliasing
Is column aliasing supported?

supportsConvert
Is the CONVERT function between SQL types supported?

supportsConvert
Is CONVERT between the given SQL types supported?

supportsCoreSQLGrammar
Is the ODBC Core SQL grammar supported?

supportsCorrelatedSubqueries
Are correlated subqueries supported? A JDBC CompliantTM driver always returns true.

supportsDataDefinitionAndDataManipulationTransactions
Are both data definition and data manipulation statements within a transaction supported?

supportsDataManipulationTransactionsOnly
Are only data manipulation statements within a transaction supported?

supportsDifferentTableCorrelationNames
If table correlation names are supported, are they restricted to be different from the names of the tables?

supportsExpressionsInOrderBy
Are expressions in "ORDER BY" lists supported?

supportsExtendedSQLGrammar
Is the ODBC Extended SQL grammar supported?

supportsFullOuterJoins
Are full nested outer joins supported?

supportsGroupBy
Is some form of "GROUP BY" clause supported?

supportsGroupByBeyondSelect

Can a "GROUP BY" clause add columns not in the SELECT provided it specifies all the columns in the SELECT?

supportsGroupByUnrelated

Can a "GROUP BY" clause use columns not in the SELECT?

supportsIntegrityEnhancementFacility

Is the SQL Integrity Enhancement Facility supported?

supportsLikeEscapeClause

Is the escape character in "LIKE" clauses supported? A JDBC CompliantTM driver always returns true.

supportsLimitedOuterJoins

Is there limited support for outer joins? (This will be true if supportFullOuterJoins is true.)

supportsMinimumSQLGrammar

Is the ODBC Minimum SQL grammar supported? All JDBC CompliantTM drivers must return true.

supportsMixedCaseIdentifiers

Does the database treat mixed case unquoted SQL identifiers as case sensitive and as a result store them in mixed case? A JDBC CompliantTM driver will always return false.

supportsMixedCaseQuotedIdentifiers

Does the database treat mixed case quoted SQL identifiers as case sensitive and as a result store them in mixed case? A JDBC CompliantTM driver will always return true.

supportsMultipleResultSets

Are multiple ResultSet from a single execute supported?

supportsMultipleTransactions

Can we have multiple transactions open at once (on different connections)?

supportsNonNullableColumns

Can columns be defined as non-nullable? A JDBC CompliantTM driver always returns true.

supportsOpenCursorsAcrossCommit

Can cursors remain open across commits?

supportsOpenCursorsAcrossRollback
Can cursors remain open across rollbacks?

supportsOpenStatementsAcrossCommit
Can statements remain open across commits?

supportsOpenStatementsAcrossRollback
Can statements remain open across rollbacks?

supportsOrderByUnrelated
Can an "ORDER BY" clause use columns not in the SELECT statement?

supportsOuterJoins
Is some form of outer join supported?

supportsPositionedDelete
Is positioned DELETE supported?

supportsPositionedUpdate
Is positioned UPDATE supported?

supportsSchemasInDataManipulation
Can a schema name be used in a data manipulation statement?

supportsSchemasInIndexDefinitions
Can a schema name be used in an index definition statement?

supportsSchemasInPrivilegeDefinitions
Can a schema name be used in a privilege definition statement?

supportsSchemasInProcedureCalls
Can a schema name be used in a procedure call statement?

supportsSchemasInTableDefinitions
Can a schema name be used in a table definition statement?

supportsSelectForUpdate
Is SELECT for UPDATE supported?

supportsStoredProcedures

Are stored procedure calls using the stored procedure escape syntax supported?

supportsSubqueriesInComparisons

Are subqueries in comparison expressions supported? A JDBC CompliantTM driver always returns true.

supportsSubqueriesInExists

Are subqueries in 'exists' expressions supported? A JDBC CompliantTM driver always returns true.

supportsSubqueriesInIns

Are subqueries in 'in' statements supported? A JDBC CompliantTM driver always returns true.

supportsSubqueriesInQuantifieds

Are subqueries in quantified expressions supported? A JDBC CompliantTM driver always returns true.

supportsTableCorrelationNames

Are table correlation names supported? A JDBC CompliantTM driver always returns true.

supportsTransactionIsolationLevel

Does this database support the given transaction isolation level?

supportsTransactions

Are transactions supported? If not, invoking the method commit is a noop and the isolation level is TRANSACTION_NONE.

supportsUnion

Is SQL UNION supported?

supportsUnionAll

Is SQL UNION ALL supported?

usesLocalFilePerTable

Does the database use a file for each table?

usesLocalFiles

Does the database store tables in a local file?