View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Directory of Open Access Journals

Database Access Through Java Technologies

Ion LUNGU, Nicolae MERCIOIU
Faculty of Cybernetics, Statistics and Economic Informatics,
Academy of Economic Studies, Bucharest, Romania,
ion.lungu@ie.ase.ro , nicu.mercioiu@gmail.com

As a high level development environment, the Java technologies offer support to the development of distributed applications, independent of the platform, providing a robust set of methods to access the databases, used to create software components on the server side, as well as on the client side. Analyzing the evolution of Java tools to access data, we notice that these tools evolved from simple methods that permitted the queries, the insertion, the update and the deletion of the data to advanced implementations such as distributed transactions, cursors and batch files.

The client-server architectures allows through JDBC (the Java Database Connectivity) the execution of SQL (Structured Query Language) instructions and the manipulation of the results in an independent and consistent manner. The JDBC API (Application Programming Interface) creates the level of abstractization needed to allow the call of SQL queries to any DBMS (Database Management System). In JDBC the native driver and the ODBC (Open Database Connectivity)-JDBC bridge and the classes and interfaces of the JDBC API will be described.

The four steps needed to build a JDBC driven application are presented briefly, emphasizing on the way each step has to be accomplished and the expected results. In each step there are evaluations on the characteristics of the database systems and the way the JDBC programming interface adapts to each one. The data types provided by SQL2 and SQL3 standards are analyzed by comparison with the Java data types, emphasizing on the discrepancies between those and the SQL types, but also the methods that allow the conversion between different types of data through the methods of the ResultSet object.

Next, starting from the metadata role and studying the Java programming interfaces that allow the query of result sets, we will describe the advanced features of the data mining with JDBC. As alternative to result sets, the Rowsets add new functionalities that enhance the flexibility of the applications. These are analyzed and the approach is described.

Keywords: *Java, JDBC, Database access, SQL*

1 Introduction

Java plays a dominant role in client-server programming, in the presentation layer of the websites, but also in the business logic on the applications servers. A large contributor to this success is attributed to the ability to interact with data. Starting from these advantages, a description of the JDBC (*Java Database Connectivity*) was needed, also the way these instruments can be used, emphasizing on the new features the latest version have to offer.

The standardization of SQL (*Structured Query Language*) did not block several DBMS creators to develop proprietary extensions to SQL, rezulting in the creation of different interfaces for data manipulation. However, JDBC technology offers a consistent interface for manipulating data, regardless of the format in which the data is stored (fig. 1).

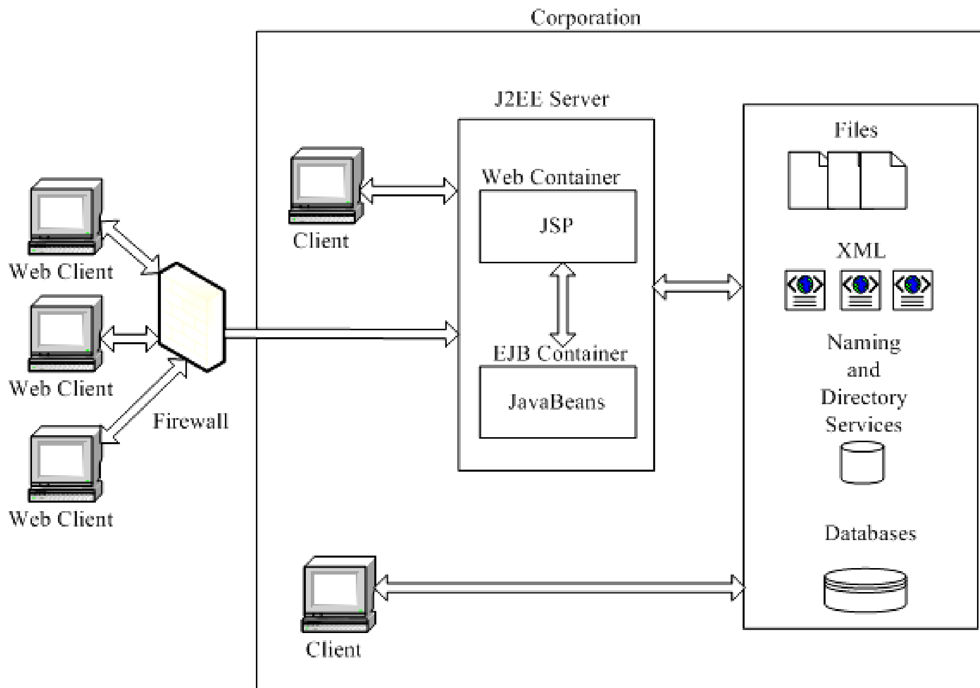

Fig. 1. The role of Java technologies to access data at enterprise level. Derived from [4] , pag. 7

2. The Evolution of Data Access Java Instruments

When Sun Microsystems released the first JDBC API 1.0 (*Application Programming Interface)* in 1997, it had several shortcomings, for instance the interface to access SQL databases. JDBC 2.0 arrived with new features such as cursors and batch files. Also, the *Optional Package*, *javax.sql* as well as other advanced features such as distributed transactions or the *RowSet* interface arrived.

JDBC 3.0 brought transactional intermediate saving points and support for SQL99 types of data. The optional packages have been included in the Java 1.4 distribution. JDBC 4.0 provides support for SQL 2003 but also extended support for CLOB (*Character Large OBject*) and BLOB (*Binary Large OBject*).

Currently, the Java API includes a JDBC-ODBC driver (*Open Database Connectivity-Java Database Connectivity*) that allows the JDBC driver access to a native system database, when an ODBC native system driver exists for that database. The Java API does not include drivers for all databases. The existence of a common programming interface brings several benefits. Otherwise, if any database creator would built its own API, that would lead to thousands of ways of programming databases, so any interface would have to be known. Luckily the software industry has chosen the JDBC standard, easying the work of developers, and allowing the creation of robust and scalable applications.

3. The Architecture

In client-server architecture the databases reside on the same or on a different machine on which the client connects to the intranet or Internet. JDBC allows to use SQL instructions and to process results of the queries in an independent and consistent manner. Using a high level of abstractization represented by the JDBC API, the situation of the programmer to handle different SQL calls to a certain DBMS (*Database Management System*) is avoided. (fig. 2).

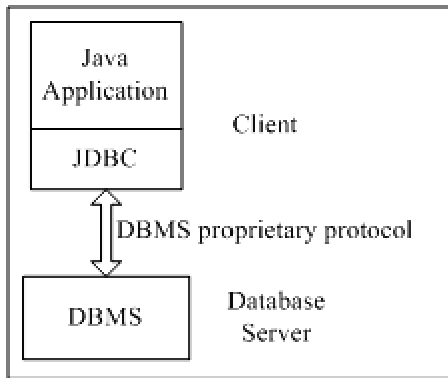

Fig. 2. The role of JDBC in accessing data –Derived from [3] , pag. 442

In order to be able to connect to a certain DBMS we only need to switch the driver, operation that can be done dynamically, even when the application runs, without the recompilation of the application.

The way those drivers are built is standardized through the JDBC specifications which describes the standard interfaces that are to be implemented. During time an evolution of those specifications occured, and the functionalities have been enhanced without compromising the compatibility with previous versions of the specifications. Generally, the JDBC specifications describe a series of interfaces that the people who develop the drivers should implement. Some databases do not allow stored procedures or other functionalities due to non-standard development of databases in general. Therefore, the JDBC specifications that have emerged from time forced the drivers creators to implement a reduced set of interfaces, other things remaining optional, without restricting the real posibilities of existing databases.

JDBC provides object-oriented access to databases through the definition of classes and interfaces that cover several abstract concepts. Also, the JDBC standard defines a series of interfaces that are to be implemented by the drivers creators in order to give the developers informations about the queried database, the DBMS used and so on. Those intels are also known as *metadata*, which means „data about data".

JDBC programming covers many aspects: client-server communication, drivers, APIs, data types and SQL instructions. The JDBC API releaves much of the burden needed to create applications with databases. It comprises many simple and intuitive components that can work for the programmer. In order to create an application one just need to assemble these components. Programming JDBC is also a very methodical way. 90% of the JDBC application uses the same objects and methods.

The first step is to obtain, install and configure the JDBC driver. Afterwards the needed component can be utilized in all the JDBC applications. After that compiling takes place, running the application and solving the eventual issues.

JDBC is an API that encapsulates calls on two levels needed to access database data and interacts through a common interface. JDK *(Java Development Kit)* and JRE (*Java Runtime Environment*) both contain the standard API, the interfaces and classes being contained in two major packages *java.sql* and *javax.sql*. The first package includes standard components, whilst the second includes enterprise level components.

The entire communication with the database occurs through the JDBC drivers. This driver converts the SQL instructions into a database server comprehensible format, using the correct networking protocols. JDBC abstracts the specific communication with the database. (fig. 3).

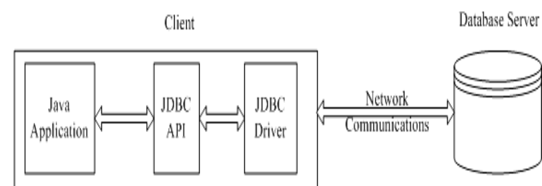

Fig. 3. The relationship between the application-JDBC-database
Derived from [4] , pag. 31

The Java SDK includes the ODBC-JDBC driver, thus allowing the access to ODBC drivers to database. Instead of accessing directly the database, JDBC “talks” to the ODBC drivers, which, in its turn communicates with the database.

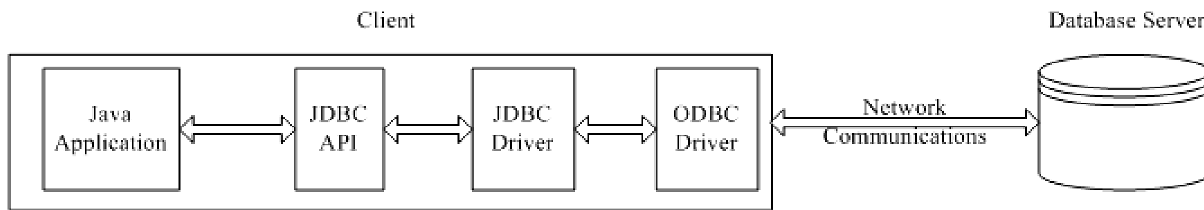

Fig. 4. The relationship application-JDBC-ODBC-database
Derived from [4] , pag. 31

Installing the JDBC driver is similar to installing any other Java API. You only have to add the path of the driver in the CLASSPATH variable when compiling and running the application. When using the ODBC-JDBC bridge driver, this step is not required, however other additional settings have to be done.

First of all, the application should be able to communicate with the database. Afterwards, the application has to be able to establish connections with the database to create a communcation channel in order to send SQL commands and retrieve results. Finallz, the application has to have a mechanism to deal with the errors. In order to accomplish all these things, the JDBC API provides the following interfaces and classes:

- *Driver* – this interface controls the communication with the database server. Rarely one should need to interact with objects of the *Driver* type. Given this, the *DriverManager* objects can be used instead. These have an abstract representation of the details associated to the work with *Driver* objects.
- *Connection* – instantiating objects of this interface represents the physical connection to the database. The result set and tranzactions can be controlled using *Connection* objects.
- *Statement* – objects created with this interface in order to send SQL commands to the database. Some derived interfaces accept supplemental parameters in order to execute stored procedures.
- *ResultSet* – these objects contain the retrieved data from the database after the query has been performed using *Statement* object. These objects allow the browsing of the data like an iterator.
- *SQLException* – a class that traps any error that is encountered in the application

Any Java application that uses databases works directly or indirectly with those four components described earlier.

4. Steps in Writing a Jdbc Application

Practically, the steps that are to be followed when writing a JDBC application are:

1. The registration of the JDBC driver with *Class.forName().newInstance()*.
2. The connection to the database is open with *DriverManager.getConnection()*.
3. A Statement type object is created in order to send SQL commands using the method *Connection.createStatement()* and afterwards *execute(), executeUpdate()* or *executeQuery()*
4. The connection is closed using the method *close()*.

The first step that is to be made in order to use a JDBC driver is the exact determination of the class for the driver provided by the creator. Usually, the producers respect the naming conventions of the packages when naming the drivers. The *java.sql.Driver* interface and the *java.sql.DriverManager* class are the tools to work with drivers. Registering a driver means the registration with a

DriverManager object. There are several techniques to register JDBC drivers:

- Class.forName(String *driverName*).newInstance()
- DriverManager.registerDriver(Driver *driverName*)
- jdbc.drivers property

In JDBC, an instance object of the type *Connection* represents a physical connection to the database. The method *Driver.connect()* can be used, being prefered though the *getConnection()* method of the DriverManager class because it allows the choose of the right driver. Also, the method can be overriden in order to allow opening of different menas o open connections. JDBC needs a special name system to be used when connecting to a database. The general format is jdbc:*<subprotocol>:<subname>* where *<subprotocol>* represents the specific protocol of the producer and *<subname>* is the source of the data (the logical name of the database we're trying to connect to).

To open connections, the *getConnection()* from the *DriverManager* class returns a valid *Connection* type object. If the method fails, *DriverManger* throws a *SQLException* containing the specific database error.

Closing the connection means the mandatory usage of the *close()* method. The method *Connection.isClosed()* does not check whether the connection is stil open or closed, but returns *true* if the method *close()* has been used. The best way to check a connection is to try a JDBC operation and the trap of the exception to determine whether the connection is still valid.

For the beginning we must be sure that the client's session has been closed on the database server. Some databases cleanse the remains if the sessions terminate unexpectadely. Then, the database sees that the user's session failed and executes a rollback to all the changes between the execution of the programme (for instance sessions that ended in the middle of the tranzaction). Explicitly closing the connections ensures that the client-server medium has been cleansed completely and makes the database administrator happier, conserving the resources used by the DBMS, for instance free licenses used on open sessions. Also RAM and CPU is spared on the server on which the database resides.

We can interact with the database in two ways. This way we can send a SQL query to obtain data about the database schema or to "learn" the values stored in some database fields, all these taking place at runtime. In that case we would need to create parametric JDBC or stored procedures. In this case we need to create. Regardless of what we want to do, the *Statement, PreparedStatemen*t and *CallableStatement* object provides the sufficient tools in order to attain our goals.

The corresponding interfaces define models and properties that allow sending commands and receiving data from and to the data database, as well as methods that help creating a bridge between different types of data defined in Java and specific to each type of SQL database types. For instance, the data types that have NULL values in the database in contrast with the *int* type in Java, or the different representation of date and time data between Java and SQL-92. There are methods that allow conversion of data from Java into JDBC. *Statement* objects offer DBMS interaction. They allow all the types of DML (*Data Manipulation Language*), DDL (*Data Definition Language*) commands to be executed, as well as other specific commands, batches and tranzaction management commands.

The three methods of the *Statement* objects are *execute(), executeUpdate()* and *executeQuery()* that allow sending commands to the database and retrieving results. *Execute()* processes DML instructuions, or DML or other specific database commands. It can return one or more *ResultSet* type objects. The method has flexibility, but the processing of the

results is a little bit difficult. *executeUpdate()* is used for INSERT, UPDATE, DELETE or DDL instructions and returns the number of records affected by the sent command. *executeQuery()* queries the database and returns a result set (a *ResultSet* object).

A *ResultSet* type object contains data returned by the SQL queries, run with one of the methods: *executeQuery()* sau *execute()*. Given the fact that many databases use a query language that have supplemental commands other than the DML or DDL, Java has support for a special type of commands format– *JDBC SQL escape* that allows access to specific functions of the database. When used this feature, the driver translates the commands in the specific format of the database. The *execute()* method is the most flexible way to interact with the database because it cas process result sets or number of records. The disadvantage here is that when used you cannot anticipate the type of the results returned – sets of results, number of records, or both.

The next figure shows the way the results returned by the *execute()* method are processed.

The interface set *Statement* can be used to process batches, i.e. sending multiple DML instructions (executed as one command) in a single call, that allows using a tranzactional control over the database. This control allows, for instance, the return to the initial state of all changes if one of the change failed, and by that insuring integrity and database consistency.

The tranzactions allow control of whether and when the changes are applied to the database. They allow the representation of a single instruction or a group of SQL instructions to be treated as a single logical unit, and if just one instruction within fails, the whole tranzaction fails. Tranzactions present both advantages and disadvantages. One advantage is that they allow consistency and integrity of the data. The disadvantage is that the blocking system for each database is different from database to database and the effect of initial blocking of the data initiated by the tranzaction can be sometimes surprising.

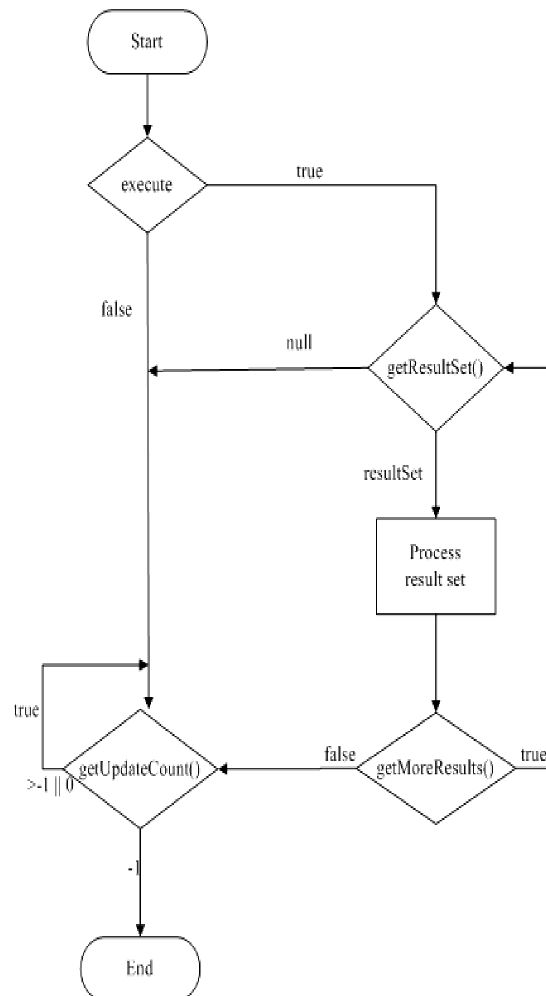

Fig. 5. The process of returning results from the execute() method - [4] , pag. 68

With JDBC tranzactions can be administered through the *Connection* type objects; for instance using the *auto-commit* module and the usage of *rollback()* method. A saving point is actually a logical tranzaction rollback point within the tranzaction. If an error occurs between the last saving point, the *rollback* method can be used to restablish the state of the data at that saving point.

The *PreparedStatement* interface offers some advantages over the classic *Statement* especially because of the feature of adding

parameters dynamically. Still, not all the databases support this feature. Also, all the commands of this type remain in memory in this open session or until the *PreparedStatement* object is closed. This *PreparedStatement* allows input and output stream, allowing us to store files in the database as values.

The *CallableStatement* object allows us to execute stored procedures in the database from the application. These objects utilize parameters as *OUT* or *INOUT*. The result sets are nothing more than rows and columns obtained from the *ResultSet* objects, creating a logical view of the data from the database. JDBC provides a class that implements the *ResultSet* interface that offers method to allow data interactivity.

Although a result set contains multiple records, at a time it is possible to get access to only one record, the "active" record. Accessing this record means moving the cursor with specific methods. Populating a set of results, the cursor initially is positioned before the first record. Obviously, in order to access data, the cursor has to be moved onto this first record. The set of records can be browsed forwards and backwards, also beyond the last records. Depending on the type of the result set, it is possible or not to go back to a previous record. If not possible, in order to access data again it is necessary to recreate the result set by executing a SQL query.

There are several types of results:

The predefined type *Standard* that allows only sequential and forward browse of the set. The data cannot be updated. It is useable to populate a simple list or other simple operations.

The second type would be *Scrollable*, which allows the browsing of the results forth and back and jumping to a specific records. This one reflects the changes in the database, so it can be used in real-time applications.

The third type would be *Updateable* that allows the update of the result set without additional SQL instructions.

The *Scrollable* and *Updateable* must be used only when really needed as they can affect the application's performance.

5. Main Types of Data

Generally, the databases support a limited types of data. If SQL2 (SQL92) offered support for limited standard types, SQL3 allows customized built types, the dimension of data that can be accomodated in a column is now bigger than 1GB of binary or character data. Also, SQL offers complex object support in business modelling and multimedia applications as well as object identifiers, abstract data and inheritance. However, not all the databases support SQL3 standard.

There are discrepancies between Java types and known database types, that requires the conversion of those Java types into SQL types and viceversa. These conversions are made through the *getXXX(), setXXX()* and *updateXXX()* methods that belong to the *ResultSet* object. It is important to know that every JDBC data type has a corresponding recommended Java type. Still, these methods are not very strict, they allow conversions from more precise into more loose types of data and even into other types (for instance: *getString()*).

Given the fact that primite Java types do not have to be defined, they store directly information, remaining constant from one application to another and from one virtual machine to another. Because primitive objects cannot be instantiated, Java offers the *wrapper* class that allows treating the primitive values as objects.

In SQL, NULL represents a data with unknown or undefined value. In Java, this NULL can present a problem, especially for numeric data types. For instance, the integer type from Java cannot have NULL values. Using the *ResultSet.getInt()* method JDBC will translate this NULL value into

0, which untreated can lead to an erroneous interpretation of the data. Objects, on the other hand, can have NULL values in Java. The *ResultSet.wasNull()* methods determines whether the last column read from the database returned a NULL value.

Data returned from SQL queries must be formatted as JDBC types. The conversion to Java types must be made before assignation to variables.

SQL UDTs (*User-Defined Types*) allows the developers to create their own definition of data types in the database. These are exlusively defined with SQL instructions, but JDBC offers support for UDT in Java applications. The custom types are materialized on the client, so the access is not directly to the value, but through an intermediate *LOCATOR* that references a value in the database. The UDTs allow the usage of large data and the way those can be used will be presented later on.

The *DISTINCT* data type allows the assignation of the new custom data type with another type of data, in a similar manner classes are extended in Java.

STRUCT is a data type built that has several members, named attributes, each of them carrying different types of data. A Java class without methods is an analogical representation of a *STRUCT*. In SQL3 *STRUCT* types of data can be constructed, each being able to hold any type of data, including other *STRUCT*.

Example: Declaring a *STRUCT:*

```
CREATE TYPE Sal_DATA(
CNP Number(9),
Nume VARCHAR(20),
Prenume VARCHAR(20),
Salariu NUMBER(9,2) )
```

JDBC allows the creation of Java classes to mirror UDTs on the database servers. The process of creation and usage of a Java class is called mapping of types. The advantages are: control of access through classes, data protection, the possibility to add new methods and attributes.

6. Data Mining With Jdbc

Understanding the concept of data mining implies the knowledge of the role of the metadata.

These are data about data. In databases, metadata represents information about data and structure and applications that deal with data. An example would be tables and attributes of the columns.

The JDBC API allows the descovery of the metadate about a database through the query of the result set using the *DatabaseMetaData* and *ResultSetMetaData* interfaces. The first one allows gathering information about the database attributes and allows taking decision at runtime upon these information. The second interface allows gathering the attributes such as number of columns, name and type of data of the result set. This information can be used, for instance, to populate a report with the name of the column and to determine which kind of *getXXX()* method should be used.

A *ResultSetMetaData* object can be used to create a generic method for processing result sets. This way, the types of the data from the column of the result set and the correct version of *getXXX()* methods to obtain data. *DatabaseMetaData* allows the creation of tools which database administrators can use to inspect databases, the structure of the tables and the users schemas.

JDBC 3.0 defines a new interface for metadata - *ParameterMetaData*. This one describes the number, type and properties of the parameters used in prepared statements.

The *ResultSetMetaData* provides information about the columns in the result set, such as number and type. The interface does not provide information about the number of records in the set of results.

The *DatabaseMetaData* interface is useful when inspecting the structure of the database. Creating a *DatabaseMetaData* object can be done by using the *getMetaData()* method of the *Connection* object. A *DatabaseMetaData* object has

many methods and properties, all those can be grouped in two categories: refering to the characteristics of the database, or refering to the structure of the database.

The first category of methods and properties answers to questions such as:

- Does the database support batches?
- What is the user which I am connected to the database?
- What kind of SQL data types does the database support?
- What are the SQL keywords supported by the database?

The methods from this category refer to information about the database return *String* results; the ones that offer information about database limitations return *int*.

The methods from the second category return a *ResultSet* object which depends on the method used to query the database. The majority of the methods are simple and allow the usage of replacement wildcards: "_" is used to replace a single character, while "%" can be used to replace zero, one or more characters.

Information pertaining to the tables and columns can be obtained only if sufficient access rights are given. It is recommended to use pattern of strings to avoid obtaining a result set that is too big and unusable. The *getUDT()* method offers information about the UDTs in the database. *getPrimaryKeys()* and *getImportedKeys()* provide information about the primary key and the foreign keys. The *getProcedures()* and *getProcedureColumns()* methods provide information about the stored procedures in the database.

7. ***Rowsets***

Rowsets represent an alternative to result sets. The *RowSet* interface extends the *ResultSet* offering the same functionalities for viewing and manipulating data, but adds among features, functionalities that enhance the flexibility and the power of the application. Rowsets implement the *JavaBean* architecture, can operate without a coninuous connection to the data source and can offer tabulary data about any data source being in constrast with result sets that can only work with databases.

Extending the *ResultSet* interface, the *RowSet* allows access to the same methods and properties. A *RowSet* object can obtain data from a source in many other ways. The main differences betwessn these two interfaces are:

- The *RowSet* interface supports the *JavaBean* component model, allowing the developers to use the visual tools for *Beans*. *RowSet* can inform the "listeners" about events that appear.

- The row sets can operate connected or disconnected. The first way is similar to the result sets, but the disconnected stores the rows and the columns in memory, allowing the manipulaton of data in this manner.

Because the *RowSet* is in the *javax.sql* package, Sun Microsystems does not provide a standard implementation. Still, in JDBC 2.0 there are some implementations like: *JdbcRowSet, CachedRowSet* and *WebRowSet*.

The development of *RowSet* object based applications implies a different technique than the one with standard components. Mainly we need a single object to implement the *RowSet* interface. The steps to be pursuit when using a row set are:

1. Registering the JDBC driver.
2. Setting the connection parameters.
3. Populating the row set.

Because the *RowSet* object supports the *JavaBean* model it is impossible to access the properties of the object directly, which requires the usage of the methods *get* and *set* to configure the properties of the class that implements the *RowSet* interface.

RowSet objects can generate *JavaBean* events and allow the notification of other components the events that appear in the *RowSet* object. A row set, acts differently than a result set, because it automatically

connects to the data source when it has to retrieve or update data.

Both DDL and DML commands can be used with *RowSet* objects but the execution is different than in standard JDBC. First of all, it is not necessary to instantiate *Statement, PreparedStatement* or *CallableStatement* objects to execute SQL instructions. The object determines if there is a parametrized query or a stored procedure to be executed.

The retrieve of the data from the row sets is done with the *getXXX()* methods, where *XXX* refers to the Java type of data in which we store the value.

Since the *RowSet* interface extends the *ResultSet* interface, for browsing the rows the same methods exposed by the *ResultSet* can be used. Also, the properties can be controlled with *scrollable* and *updateable* by the *setType()* method.

After usage, the *RowSet* object must be closed in order to free the database resources used. The *RowSet.close()* method frees all the resources of the database. Closing the object is critical when this is of *JdbcRowSet* type, because this object maintains an open connection to the server once the *execute()* method is called. Objects of type *CachedRowSet* and *WebRowSet* connect to the data source when needed. However to eliminate the possibility of unwanted closing by the garbage collector, they must be explicitly closed.

Using a *JdbcRowSet* object is simple because it is a *JavaBean* component. Once the row set is populated, the methods inherited from *ResultSet* can be used to work on data. This object does not require a JDBC driver, or an open connection to the database.

The *CachedRowSet* object provides a disconnected and serializable implementation of the *RowSet* interface. Once the object is populated, it can be made serialized so we can share information with other users. It is not recommended for large amounts of data since it can exhaust the system memory.

The *WebRowSet* can work independent and is able to serialize data. This object is able to generate an XML (*eXtensible Markup Language*) file which can be used as it is or use an XML file to repopulate itself. Having a row set represented as an XML file, the data can be presented on various devices and browsers. The *WebRowSet* class works great with HTTP (*HyperText Transfer Protocol*). The client and the server exchange XML documents that represent *WebRowSet* objects. The construction of such an object is not trivial, but the browsing and he manipulation of the data set is done in a similat manner to the other types of objects presented earlier.

Conclusions

The designers of IT systems choose the combination of Java and JDBC because it allows the disemination of the information contained within databases in a simple and economic way. The operations within the organization can go on by utilizing existing databases even if these are used on different operating systems. The time used to develop new applications is shorter and the installation and versioning control is simplified. All these advantages determined us to try to describe in a non-exhaustive manner the concepts, the methods and the techniques related to JDBC technology to access databases, offered by the Java platform from Sun Microsystems.

References

[1] Leția T., Programare *avansată în Java*, Editura Albastră, 2002;

[2] Patel P., Java *Database Programming with JDBC*, The Coriolis Group, 1996;

[3] Tanasă Ş., Olaru C., Andrei Ş., *Java de la 0 la expert*, Polirom, 2003;

[4] Thomas T., Java *Data Access JDBC, JNDI, and JAXP*, M&T Books, 2002;

[5] Văduva C., *Programare în Java*, Editura Albastră, 2002;

[6] Sun Microsystems, *JDBC Data Access API*, http://java.sun.com/products/jdbc;