Generic Object-Oriented Database System

Konstantin Knizhnik

March 2, 1999

Contents

List of Tables

1 Introduction

JORA will simplify you development of database application, establishing implicit mapping between relational database tables and Java classes. JORA is using JDBC for database connection, so it can be used with any database supporting JDBC or ODBC. JORA also allows you to use object-oriented features, such as polymorphism and complex objects, in your database applications.

2 How it works

Programmer should create Table object for each class, which should be mapped to database table. Usually name of the class is the same as the name of database table. But it is possible to map several classes with different names to one database table. Also names of class components should match names of table columns (it is possible for class to contain not all fields from the database table, so class can be considered as some kind of table view). A component of mapped on table class should be either of one the types, supported by JDBC (see table Atomic types), or object with such components, or object implementing java.io.Serializable interface. In second case, database table should contain all fields from classes of object compoentns. If class component is object implementing Serializable interface, then database table should contain large binary column for storing contents of objects closure. Consider the following Java classes:

```
    public class Point {
        public int x;
        public int y;

public boolean belongs(int left, int top, int width, int height) {
    return x >= left && x <= left + width &&
```

```
           y >= top && y <= top + height)
}
public static String sqlBelongs(String name,
   int left, int top, int width, int height)
{
    return name + "_x BETWEEN " + left + " AND " + (left + width) +
     " AND " + name + "_y BETWEEN " + top + " AND " + (top + height);
}
    }

    public class R2Object {
        public int    id;
        public String name;
        public Point  location;

        public static Table table = new Table("R2Object", null, session, "id");
    }
    public class Circle extends R2Object {
        public int    radius;
        public static Table table = new Table("Circle");
    }
    public class Line extends R2Object {
        public Point  lineTo;
        public static Table table = new Table("Line");
    }
```

And database containing correspondent tables:

```
    create table Circle(
        id         SMALLINT PRIMARY KEY,
name       CHAR(10),
location_x INTEGER,
location_y INTEGER,
radius     INTEGER
    );
    create table Line(
        id         SMALLINT PRIMARY KEY,
name       CHAR(10),
location_x INTEGER,
location_y INTEGER,
lineTo_x   INTEGER,
lineTo_y   INTEGER
    );
```

Now it is possible to extract all records from Circle table using select method of the Table class:

```
    Cursor c = Circle.table.select("order by name");
    Detail d;
    while ((d = (Detail)c.next()) != null) {
        System.out.println("("+d.location.x+","+d.location.y+") - " + d.id);
    }
```

Or perform polymorphic query and select records from all these tables using selectAll method of the Table class:

```
    Object selection[] =
```

```
      R2Object.table.selectAll
        ("where " + Point.sqlBelongs("location", 0, 0, 100, 100)).toArray();
    for (int i = 0; i < selection.length; i++) {
        System.out.println(((R2Object)selection[i]).name);
    }
```

If you have large number of tightly linked objects, it is often very inconvenient and inefficient to store all these objects in tables of relational database. JORA allows you to pack the closure of such objects and store it as BLOB field in database. Lets look at the following example:

```
    public class Detail implement java.io.Serialized {
        public int      id;
        public String   name;
        public Detail[] connectedWith;

public Detail(int id, String name, int nLinks) {
    this.id = id;
    this.name = name;
    this.connectedWith = new Detail[nLinks];
}
    }

    public class Assembly {
        public String name;
public double weight;
public Detail details;

public static Table table = new Table("Assembly", session, "name");
    }
```

Objects of Assembly class can be stored in the following table:

```
    create table Assembly(
       name    VARCHAR(255) PRIMARY KEY,
       weight  REAL,
       details LONG RAW
    );
```

Such object cluster representing an assembly can be stored and retrieved from database without any explicit efforts of programmer:

```
    Assembly asm = new Assembly();
    asm.name = "Z-195";
    asm.weight = 30.5;
    Detail bolt = new Detail(105, "bolt", 1);
    Detail screw = new Detail(106, "screw", 1);
    asm.details = bolt;
    bolt.connectedWith[0] = screw;
    screw.connectedWith[0] = bolt;

    Assembly.table.insert(asm);
    asm.weight += 1.1;
    Assembly.table.update(asm);

    asm = (Assembly)Assembly.table.select("where name='Y-315'");
    System.out.println(asm.details.name);
```

3 Selecting objects from database

JORA supports two kinds of object queries from database: simple query, when object are extracted only from one database table, and polymorphic query, when objects are also selected from all database tables, which are derived from the specified table. Inheritance relationship between tables is specified by correspondent relationship between Java classes mapped to the table. JORA also supports notion of abstract table, which is not present in database and is used only for issuing polymorphic queries (table R2Object from the example above). Abstract tables are created by constructor with tableName parameter equal to null.

When Table object is constructed, you should pass to constructor name of Java class, name of database table, session object and name of primary key for this table. If table name is the same as class name without package prefix, then it is possible to specify only the class name: JORA provides special constructor for this case. Session object is used to encapsulate connection with database server. It should be opened before first access to the table. Primary key is used for update and remove operation, to locate necessary record within table. Primary key should be atomic. It is possible not to specify session and primary key parameters in derived tables, they will be taken from base table.

Important! All non-atomic components of mapped on database table classes should have constructor with no parameters (default constructor) to make it possible for JORA to create instances of this object when data is loaded from database. Otherwise NoSuchMethodError error will be raised.

JDK 1.1 restriction Prior to version 1.2 of JDK, it is not possible to access through reflection mechanism fields of non-public classes from other packages. That is why all classes, mapped on database tables, should be defined as public (within package) and their components also should be public, otherwise JORA package will be not able to store and extract component values (IllegalAccessError will be raised). There is no such restriction with JDK 1.2.

JORA automatically constructs list of fields, which should be fetched from the database table, creates new object instances and assigns extracted vales to object components. Method used to retrieve data from the database server depends on the type of Java object component. For example, if component has int type, then getInt() method is used. If component type is java.io.InputStream, then method getBinaryStream is used for peeking value of this column. If component has non built-in type (such as int, char, float...), then extracting of NULL values is handled by assigning null to this object component.

Important! Different SQL databases support different sets of basic types. For examples types DATE, TIME, TIMESTAMP, which are defined in SQL-92 standard, are implemented by only a very small subset of the major databases. In some cases JDBC driver failed to map JDBC type at the type supported by concrete database (for example, with MS SQL 6.5 and JDBC-ODBC bridge, method java.sql.PreparedStatement.setDate() throws "Driver not capable" exception). JORA tries to fix this problem by doing conversion of the object to string and allowing database server to perform necessary conversion from string to target database type. There is also a problem with choosing types for table columns when database table is created. Different vendors use different names for the same type. For example most of database supports BLOBs, but in Oracle it will be named "LONG RAW", in MS SQL and Sybase - "IMAGE", "BYTE" in Informix and `LONG VARCHAR FOR BIT DATA` in DB2. Unfortunately JORA can not help you in solving this problem and creating portable applications, working with different databases. You should either choose most standard types (such as INTEGER, CHAR, VARCHAR) or tune your application for concrete database.

JORA can return set of selected objects in two ways: as an array of objects or by means of Cursor. Cursor allows you to successively access selected objects. As far as polymorphic queries can require issuing of more than one SQL statement to fetch data from different tables, it is also task of the Cursor class to construct necessary SQL statements for each such table. Extracting selected objects into array makes it possible to perform random access to selected objects. It is possible to limit size of the array, so that only restricted number of objects will be placed in the array. Objects placed into the array should have no components with "lazy" extraction, such as InputStream or Blob, because they will be closed after fetching of the next record. Attention! Because of polymorphic select is represented by several SQL statements, sorting directives will not work correctly in this case (records will be ordered only locally within each table).

Cursor can be also used for updating or deleting current records pointed by cursor. In this case clause "for update" should be added kin select stattement: Assembly.table.select("for update"); Not all database drivers support such operations with cursor. So this methods will no work with such database drivers. Alternative way for execution update/remove operations is to use primary key to locate affected record.

JORA also supports "Query By Example" approach for selecting objects from database. In this case, you do not need to explicitly specify select condition, but instead of it pass object, which non-null components are used as search criteria. More precisely, components of the object with non built-in type (int, char,...) are check for null, and components with non-null values are added to search condition. The result of such query will be a set of objects, having the same values of components from the search list as object used as example. It is convenient to use this approach when user should fill some request form and then application shows list of objects matching specified criteria. Also using Query by Example approach can cause better performance, because in this case JORA caches prepared SQL statements and database server have not to parse them each time again. JORA uses hash table to search for previously prepared statements.

As far as it was mentioned above, JORA supports complex object, which fields are automatically scattered when object is constructed from database tables values, and gathered when record is inserted or updated. Such complex object can be also included in search condition, which is formed by programmer. But JORA can't help you in this case. Suggested approach, which allows you to preserve data encapsulation, is to add to the class special method, which can be used to construct query. For example, if we have Complex class, which consists of two components, we can define the following method for constructing search condition:

```
    public class Complex {
        public double re;
public double im;

       /** Normal comparison method
        */
public boolean equal(Complex c) {
    return c.re == re && c.im == im;
}
       /** Generate condition for SQL statement
        */
        public String sqlEqual(String name, Complex c) {
    return name+"_re="+c.re+" AND "+name+"_im="+c.im;
        }
    }
```

Consider the following class with component of Complex class:

```
    public class Resistor {
        public int     id;
        public String  name;
        public Complex resistance;

        public Static Table table = new Table("Resistor", session, "id");
    }
```

The definition of correspondent database table can be:

```
    create Table Resistor(
        id             INTEGER PRIMARY KEY,
        name           VARCHAR(80),
        resistance_re REAL,
        resistance_im REAL
    );
```

byte	Byte	String
short	Short	char[]
int	Integer	java.sql.Date
long	Long	java.sql.Time
float	Float	java.sql.Timestamp
double	Double	java.math.BigDecimal
boolean	Boolean	java.io.InputStream

Table 1: Atomic types for components of classes mapped on database

Symbol '_' is used as separator of field names of compound objects. This symbol is defined in class Table and can be changed to any other symbol. The query of Resistor objects can be constructed in the following way:

```
public Resistor findResistor(Complex resistance) {
    return Table.Resistor.select("where " +
                                 Resistor.sqlEqual("resistance",
                   resistance));
}
```

This solution is not so elegant, but at least it makes it possible to use such approaches as polymorphism and encapsulation. See also example in previous paragraph with Point object.

4 Insert, update and delete commands

It is possible to insert new record in the table using Table.insert(Object obj) method. Values of componets of specified object will be stored in correspondent columns of new database record. If some object component is null, then value of record column will be NULL. In case of components of Serializable type, the closure of referenced objects (i.e. object referenced by this object, objects referenced from objects referenced by this objects,...) will be packed and placed in database BLOB field using standard Java serialization mechanism.

Update and insert commands can be performed either by using table primary key or cursor current object. In the first case affected record is determined by primary key field of the object. Located record is either deleted or updated with values of components of specified object. JORA supports only keys of atomic types, if you need to update record in table with compound key, then use cursor. In the second case record located by current cursor position is used. It is not necessary to specify object in this case, because current object of Cursor class is used for updating current record. Attention! Cursors are not supported by all databases.

5 Release notes

JORA is distributed as separate Java package together with documentation prepared by javadoc Package-jora.html and test program containing classes Test.java, Employee.java, Programmer.java, Manager.java, Experience.java and Activity.java. Special script compile can be used to compile JORA package and test program and create HTML documentation. This test program was tuned for Microsoft SQL server and is using default account "sa" with empty password. So you may need to change in this test driver, database URL, user name and password according to concrete database server your are using. Also names of SQL types, which are used for table creation, are candidates for replacing. Test can be executed by running java Test command.

JORA is freeware and is distributed with sources and without any restrictions. E-mail support is guaranteed. I will do my best to fix all reported bugs and extend JORA functionality. Any suggestions and comments are welcome. Version of Jora for JDBC 2.0, supporting batch statement execution, efficient update and delete for current record, Blob and Clob data types, will be available as soon as implementation of JDBC 2.0 will be ready.