99-UJ

f-v.

Proceedings

Technology of Object-Oriented Languages and Systems

TOOLS 30

August 1-5, 1999

Santa Barbara, California

Edited by

Donald Firesmith, Richard Riehle, Gilda Pour, and Bertrand Meyer

Sponsored by

Interactive Software Engineering, Inc.

IEEE COMPUTER SOCIETY

Los Alamitos, California

Washington • Brussels • Tokyo

Using Java to add "Stored Procedures" to Databases

Raimund K. Ege, Naphtali Rishe, Jingyu Liu, Vladimir Lebedev
High Performance Database Research Center *
School of Computer Science, Florida International University
University Park, Miami, FL 33199
ege@cs.fiu.edu

Abstract

The paper describes our approach to adding "stored procedure" capability to a semantic database system using Java byte-codes and Java's ability to dynamically load and execute Java code. Several steps were necessary: first we added a Java application programmer interface to the database system; then we created a database schema to hold Java executable code; then we constructed a Java class loader to allow code to be loaded from the database; then we enabled the creation of Java objects and executed the Java code for them. Our approach is not specific to our semantic database system, rather it can serve as a recipe for adding "stored procedures" to any database system.

1. Introduction

While database systems are meant to store data, increasingly demands arise to allow data manipulation within the database context. The database system typically enables such manipulation by storing procedures [Urman, 1996, Ranking, 1997], i.e. fragments of code, that can execute with data from the database.

Object-oriented database systems [Kim and Lochovsky, 1989, Atkinson et al., 1989] - by their very definition - already deal with procedurality. While an object stored in a database represents data, it is also an instance of a class, which may define applicable methods. These methods - if made available to the database system - can be executed with data, i.e. the objects, from the database, giving the database management system procedural capabilities.

In this paper we describe our approach to add "stored procedures" capability to a database system. Our specific database system uses the semantic modeling approach [Rishe, 1992] which extends the relational database model with user definable types, explicit relations, and inheritance. The semantic database model is close to an object-oriented model, however, it lacks the capability to capture methods.

In order to add procedure capability we considered 2 approaches: (1) to define our own programming language, or (2) use an existing one. Obviously, for choice (1) we would have to specify our own language, provide grammar, parser, compiler, run-time environment in order to make it viable. The existence of the Java programming language [Gosling et al., 1996, Jaworksi, 1996] made our choice (2) easy since several of its features greatly simplify our endeavor:

1. Java code is compiled and stored on a per-class basis. One Java class yields one Java compiled-code file (byte-code file).
2. Java code can be loaded dynamically. It does not have to be linked, neither statically nor dynamically at compile time.
3. Java supports reflective programming, i.e. a running Java program can modify itself while running. For example, a Java program can load a class, inquire about its methods, create instances of the new class, and invoke those methods for these instances, all in the same program.
4. Java is platform independent, so operating system dependencies are not present.
5. Java can easily interact with the C++ programming language, which is very important since our database management system is written entirely in C++ and has a C++ interface.

In the following sections we will first outline the basic elements of the semantic data model used in this database system: we will illustrate its C++ application programmer interface (API). Then we discuss the implementation of our Java to C++ bridge, which gives us a Java API to the database system. Next we illustrate a database schema to store Java code and give an example of how to store and retrieve it. And finally we show how to execute Java stored procedures with database data.

The paper concludes with some performance results and our vision of how to further enhance the database system.

2. The Semantic Binary Database System

The target of our research effort, the semantic binary database system [Rishe, 1992], models data in a semantic fashion, but employs a highly efficient binary storage model [Rishe, 1996].

The semantic model allows the definition of categories, relations and database types. Database types exist for the most commonly found types such as numbers (arbitrary varying precision and magnitude), strings, large binary objects etc.. A category is a specification of a class of abstract objects in a database. Each category may have relations with other categories or database types. A relation from a category to a database type is called an attribute. A relation from a category to a category is called an abstract relation. Categories can have sub-categories, which models inheritance.

The storage model consists solely of binary facts, each of which describe an aspect of an abstract object in the database, as well as inverses of these facts constructed in a way guaranteeing optimality of the so-called "basic queries". Examples of such binary facts are:

- an abstract object is an instance of a certain category;
- an abstract object has an attribute with a certain value;
- an abstract object has an abstract relation to a certain category;

More details on this storage model can be found in [Rishe, 1996].

* This research was supported in part by NASA (under grants NAGW-4080, NAG5-5095, NAS5-97222, and NAG5-6830), NSF (CDA-9711582, IRI-9409661, HRD-9707076, and ANI-9876409), ARO (DAAH04-96-1-0049 and DAAH04-96-1-0278), AFRL (F30602-98-C-0037), BMDO (F49620-98-1-0130 and DAAH04-0024) DoI (CA-5280-4-9044), and State of Florida.

```
#include <iostream.h>
#include <sdb3.h>
main() {
  TDataBase* DB = OpenDataBase("Demo");
  DB->Transaction_Begin();
  // create class with attributes
  Category *cat = NewCategory(DB, "Person");
  NewRelation(DB, "name",  "Person", "String");
  NewRelation(DB, "birthYear","Person", "integer");
  // make instances
  Var p1 = DB->NewAbstract("Person");
  p1.Assign("Person::name","John Doe");
  p1.Assign("Person::birthYear",1958);
  Var p2 = DB->NewAbstract("Person");
  p2.Assign("Person::name","Sue Miller");
  p2.Assign("Person::birthYear",1965);
  // simple query
  SetQuery list = cat->GetObjects();
  Var person;
  while (list.GetVarInc(person)) {
    cout << pChar(person.Query("Person::name").GetVar()) << ", ";
    cout << pChar(person.Query("Person::birthYear").GetVar()) << endl;
  }
  DB->Transaction_End();
  CloseDataBase(DB);
}
```

Figure 1. C++ API Example

The primary application programming interface (API) to this database system is for the C++ programming language[SDB, 1995]. Figure 1 shows a simple C++ example: after opening a "Demo" database and starting a new transaction, we create a category Person, with attributes "name" of type String, "birthYear" of type integer. Then we create 2 instances of category Person and relate them to attribute values. A simple query retrieves the 2 instances from the database and prints their attribute values. The program ends with committing the transaction and closing the database.

The current C++ interface has no capability to define and attach methods to a category. In this example, a method to calculate the age of a person based on the birth year would be useful. In the following sections we will show how Java can be used to make this happen.

3. A Java to C++ bridge

The first step necessary was to create a Java application programmer interface to the semantic binary database. Our approach provides that by mimicking the C++ API with Java using the Java Native Interface [JNI, 1997] capability. We implemented a Java package that contains the same classes and functions as can be found in the C++ API. The Java classes define "native methods" which are implemented in C++ and call the respective C++ API functions.

For example, the C++ API has a function to create a new category: `NewCategory()`. The Java API defines a class `Proc` with static methods, one of them is `newCategory()`.

Figure 2 shows a portion of the Java class [1]. Other methods that were also used in the C++ example (Figure 1), such as `OpenDataBase()` and `NewRelation()` now appear as methods of class `Proc`.

```
package JavaSDBAPIBridge;
 public final class Proc extends DataBaseObject {  ...
  public native static TDataBase createDataBase( String databasename );
  public native static TDataBase openDataBase( String databasename );
  public native static void closeDataBase( TDataBase db );
  public native static Category newCategory( TDataBase db, String name );
  public native static Relation newRelation( TDataBase db,
                                             String relation,
                                             String CategoryFrom,
                                             String CategoryTo );
 ... }
```

Figure 2. Java API Proc Class

Figure 3 shows a portion of the Java class `Category`, which mirrors the C++ `Category` class. For example, the method `GetObjects()` (see Figure 1) now becomes a native method of Java class `Category`. In order to complete the Java package we provide native C++ code that is dynamically loaded whenever the Java class executes. For each native Java method we have a C++ function which simply calls the respective C++ API function. For example, Java native method `Category.getObjects()` is linked to the C++ function

```
Java_JavaSDBAPIBridge_Category_getObjects__
```

which in turn calls the `Category::GetObjects()` function in the C++ API. The standard Java developer's kit (JDK) provides helper utilities, such as the `javah` command, to facilitate the native method interface. The name conventions and further detail can be found in [JNI, 1997]. Since C++ is not platform independent we provide compiled versions for

```
package JavaSDBAPIBridge;
public class Category extends DataBaseObject {  ...
  public native String getName( );
  public native Category[] getSupercategories( );
  public native Category[] getSubcategories( );
  public native Relation[] getRelations( );
  public native boolean hasRelation( Relation relation );
  public native boolean hasSupercategory( Category category );
  public native SetQuery getObjects( );
  ... }
```

Figure 3. Java API Class Category

both environments on which the semantic binary database system is available: Sun Solaris and Windows NT.

With the Java API, we can now rewrite our earlier C++ example. Figure 4 shows the resulting Java code. The Java code quite closely resembles the C++ code of Figure 1: we

[1]DataBaseObject is a common superclass to serveral classes in the Java API. It contains operations and attributes that are common to all of its subclasses. SetQuery is a class to capture query results.

```
import JavaSDBAPIBridge.*;
import lang.io.*;.*;
public class Main {
  public static void main( String[] args ) {
    TDataBase DB = Proc.openDataBase("Demo");
    DB.transactionBegin( );
    // create class with attributes
    Category cat = Proc.newCategory( DB, "Person" );
    Proc.newRelation(DB, "name", "Person", "String" );
    Proc.newRelation(DB, "birthYear", "Person", "integer" );
    // make instances
    Var p1 = DB.newAbstract("Person");
    p1.assign("Person::name","John Doe");
    p1.assign("Person::birthYear",1958);
    Var p2 = DB.newAbstract("Person");
    p2.assign("Person::name","Sue Miller");
    p2.assign("Person::birthYear",1965);
    // simple query
    SetQuery list = cat.getObjects();
    Var person;
    while (list.getVarInc(person)) {
     System.out.println(Proc.pChar(person.query("Person::name").getVar())
       + ", " + proc.toLong(person.query("Person::birthYear").getVar()));
    }
    DB.transactionEnd( );
    Proc.closeDataBase(DB);
  } }
```

Figure 4. Java API Example Program

open the database, start a transaction, create a class, make instances, issue a simple query, and then commit the transaction and close the database.

The Java program accesses the semantic database as easily as C++.

4. The Java Class Repository

Java source code resides in files that have the extension "`.java`". While a source code file may contain more than one Java class, once it is compiled it will result in exactly one file with extension "`.class`" for each compiled Java class. Java "`.class`" files actually contain Java byte codes, which are executed in the context of a Java virtual machine [Lindholm and Yellin, 1996]. Java virtual machine implementations are available for most modern computing platforms and also are contained within typical Internet browsers. This is the key to Java's platform independence.

Figure 5 shows a simple Java class for our Person category example. It declares two attributes "name" and "birtYear", a constructor, and a simple method for age calculation. This class is quite similar to the `Person` category of Figures 1 and 4. When compiled, its code resides in file "`Person.class`".

To store Java code in the database all we need to handle is Java "`.class`" files. We create a simple category "JavaClassRepository" with attributes "name" and "data". The "data" attribute is of type "binary" which allows to store an arbitrary-long chunk of data.

```
public class Person {
   String name;
   int birthYear;
   public Person(String n, int b) {
      name = n; birthYear = b;
   }
   public int getAge(int when) { return when - birthYear; }
}
```

Figure 5. Java Person Class

```
public static void Store( String[] args ) {
  TDataBase DB = Proc.openDataBase("Demo");
  DB.transactionBegin( );
  // create classes with attributes
  Category cat = Proc.newCategory( DB, "JavaClassRepository" );
  Proc.newRelation(DB, "name", "JavaClassRepository", "string" );
  Proc.newRelation(DB, "data", "JavaClassRepository", "binary" );
  // code to load Person.class into buffer
  File theFile = new File("Person.class");
  DataInputStream fileStream =
       new DataInputStream(new FileInputStream(theFile));
  // allocate a buffer and read the data into it
  byte[] buffer = new byte[theFile.length];
  fileStream.readFully(buffer);
  // make instance
  Var c = DB.newAbstract("JavaClassRepository");
  c.assign("JavaClassRepository::name","Person");
  c.assign("JavaClassRepository::data", buffer);
  DB.transactionEnd( );
  Proc.closeDataBase(DB);
}
```

Figure 6. Storing Class Code

Figure 6 shows the Java program [2] which creates the schema and loads the Person class into the database: it first opens the "Demo" database, starts a new transaction, then makes a new category `JavaClassRepository` with two relations to attributes "name" and "data". Then it opens and reads the "Person.class" file into a buffer, and finally creates a database object and assigns its name and data. Figure 7 shows how to retrieve a Java class from the database: the code is quite similar to the code in Figure 4. After opening the database, creatin a transaction, we read all classes in the database, then commit and closae the database.

5. Execution of Java Code

And finally we want to execute Java code. Assuming that the database contains Java class code and also instances for the same class, we need to do the following steps:

1. read Java "`.class`" code from the database;

[2] for clarity, we have ommitted the Java housekeeping and exception handling code.

```
public static void Show( String[] args ) {
  TDataBase DB = Proc.openDataBase("Demo");
  DB.transactionBegin( );
  // find category, read instances
  Category cat = DB.findCategory("JavaClassRepository");
  Relation rel = DB.findRelation("JavaClassRepository::name");
  // show the list of all java classes in the database
  SetQuery list = cat.getObjects();
  Var aClass;
  while(list.getVarInc(aClass) {
    System.out.println(Proc.pChar(aClass.query(rel).getVar()));
  }
  DB.transactionEnd( );
  Proc.closeDataBase(DB);
}
```

Figure 7. Reading Class Code

2. dynamically create a Java class from "`.class`" code;
3. find instances of a corresponding category in the database;
4. create instance of Java class with data from the database;
5. execute a method.

In order to dynamically create a class in Java we need to create a special class loader. The Java code for a DBClassLoader is shown in Figure 8. It is defined as a subclass of Java class `ClassLoader` which is part of the Java standard distribution. The critical method is `loadClass` which first tries to find the class in a local cache – to prevent multiple definitions –, checks whether the requested class is a system class, and then finally calls `defineClass` which creates a new Java class based on the btye-codes passed in a buffer parameter. The next step is to resolve the class, i.e. to make sure that all other Java names used within the new class are also loaded and present.

With this `DBClassLoader` class we can now show an example that illustrates these steps. The Java code in Figures 9 and 10 first opens that database and starts a new transaction; it then finds the `JavaClassRepository` category with its attribute relations; it then reads the Java byte-codes from the database, creates an instance of the `DBClassLoader` class, and loads the byte-codes using the `loadClass` method; it then finds the constructor and the `getAge` method using Java's reflective programming capabilities. Figure 10 then continues with the code find and read `Person` data from the database, and finally creates Java `Person` objects and executes the `getAge()` method.

In the example we loaded only a single isolated class from the database. The Person class only references builtin data types "String" and "int". More complicated classes, of course, will be related to other classes: those classes can either be also fetched from the database or resolved from the local execution environment subject to the normal Java "CLASSPATH" search rules.

The performance of executing "stored procedures" was quite acceptable, posing little overhead in addition to the regular database access times. However, even that can easily be improved. For example, to execute the method "getAge()" we had to construct a Java Person object. As easily we can define a static method "calculateAge()" as shown in Figure 11 and call it instead of "getAge()". Since we do not have the overhead of creating a Java Person Object, the performance is significantly improved.

```
import java.io.*;
import java.util.*;
public class DBClassLoader extends ClassLoader {
  private byte buf[] = null;
  private Hashtable sdbClassCache = new Hashtable();
  public DBClassLoader(byte inBuf[]) {
    buf = inBuf;
  }
  public synchronized Class loadClass(String name, boolean resolve)
         throws Class NotFoundException {
    try {
      // try to find the class in cache
      Class aClass = (Class)sdbClassCache.get(name);
      if (aClass == null) {
        try {
          // check if it's a system class
          aClass = findSystemClass(name);
        } catch (Exception e) {
          System.out.println("  System class: " + name + " not found");
        }
      }
      if (aClass == null) {
        aClass = defineClass(name, buf, 0, buf.length);
        sdbClassCache.put(name, aClass);
      }
      if (resolve) resolveClass(aClass);
      return aClass;
    } catch (Exception e) {
      System.out.println("DBClassLoader.loadClass: " + e.getMessage());
      throw new ClassNotFoundException(e.getMessage());
    }
  }
}
```

Figure 8. Database Class Loader

We measured the performance by loading 20000 objects from the database: with method invocation per `Person` object the program ran for 11.4 seconds; with static method invocation without creating Java `Person` objects it ran for 10.5 seconds. Without any method invocation, just reading 20000 objects using the Java API, the program ran for 10.1 seconds.

6. Conclusion

Our approach to adding "stored procedure" capability to the semantic binary database system uses the advanced features of the Java programming language. We make it possible to store and retrieve Java code from the database and then immediately execute it with acceptable performance. Our next research step is to use this facility to allow the definition of virtual categories. Virtual categories have attributes that are computed as needed by a query.

The approach described in this paper is actually a recipe that can easily be adopted and applied to other database system that need the capability to execute stored procedures.

```
import JavaSDBAPIBridge.*;
import java.io.*;
import java.lang.reflect.*;
import PersonClassLoader;
public class Execute {
 public static void main( String[] args ) {
    TDataBase DB = Proc.openDataBase("Demo");
    DB.transactionBegin( );
    // find Repository category and it's relations
    Category cat = DB.findCategory("JavaClassRepository");
    Relation className = DB.findRelation("JavaClassRepository::name");
    Relation classData = DB.findRelation("JavaClassRepository::data");
    // retrieve Java class 'Person' from category JavaClass_Repository,
    SetQuery query = Proc.rangeQuery(className, new Var("Person"));
    query.goFirst();
    Var dataClass = new Var(query); // assuming it's not null
    Var aData = dataClass.operatorDot(classData);
    // get the length of java class data
    int expectedBytes = (int)Proc.dbfilelength(aData);
    byte buffer[] = new byte[(int)expectedBytes];
    int readBytes = Proc.dbread(buffer, (short)expectedBytes, aData);
    Proc.dbclose(aData);
    // init class loader
    DBClassLoader classLoader = new DBClassLoader(buffer);
    // load the 'Person' class from the memory buffer
    Class personClass = classLoader.loadClass("Person", true);
    // init 'Person' constructor
    Class[] paramList = { Class.forName("java.lang.String"),
                          Class.forName("java.lang.Integer") };
    Constructor personConstructor = personClass.getConstructor(paramList);
    // init 'getAge' method
    Class[] whenList = {Class.forName("java.lang.Integer")};
    Method ageMethod = personClass.getMethod("getAge", whenList);
```

Figure 9. Executing Java Class Code

References

[Atkinson et al., 1989] Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., and Zdonik, S. (1989). The object-oriented database system manifesto. In *Proceedings of the First Conference on Deductive and Object-Oriented Databases*, Kyoto, Japan.

[Gosling et al., 1996] Gosling, J., Joy, B., and Steele, G. (1996). *The Java Language Specification.* Addison Wesley.

[Jaworksi, 1996] Jaworksi, J. (1996). *Java Developer's Guide.* Sams Net, Indiana.

[JNI, 1997] JNI (1997). *Java Native Interface Specification.* JavaSoft, A Sun Microsystems, Inc. Business, http://www.javasoft.com/products /jdk/1.1/docs/guide/jni/spec/jniTOC.doc.html.

[Kim and Lochovsky, 1989] Kim, W. and Lochovsky, F. H., editors (1989). *Object-Oriented Concepts, Databases and Applications.* ACM Press, Reading, Mass.

[Lindholm and Yellin, 1996] Lindholm, T. and Yellin, F. (1996). *The Java Virtual Machine Specification.* Addison Wesley.

```
    // Find category, relations which represent class 'Person' in database
    Category cPerson = DB.findCategory("Person");
    Relation rPersonName = DB.findRelation("Person::name");
    Relation rPersonBYear = DB.findRelation("Person::byear");
    // Retrieve all instances of class 'Person' from database
    SetQuery list = cPerson.getObjects();
    Var aClass;
    while (list.getVarInc(aClass)) {
      // retrieve data for each person object
      String name = Proc.pChar(aClass.query(rPersonName).getVar());
      int byear = Proc.toLong(aClass.query(rPersonBYear).getVar());
      Object[] paramList = {new String(name), new Integer(byear)};
      // create an instance and invoke method
      Object obj = personConstructor.newInstance(paramList);
      System.out.println(name+" is "+ageMethod.invoke(obj, 1998)+ " old");
    }
    DB.transactionEnd( );
    Proc.closeDataBase(DB);
 }
}
```

Figure 10. Executing Java Class Code (continued)

```
public class Person {
   String name;
   int birthYear;
   public Person(String n, int b) {
      name = n;
      birthYear = b;
   }
   public static int calculateAge(int when, int birthYear) [
      return when - birthYear;
   }
}
```

Figure 11. Java Person Class

[Ranking, 1997] Ranking, R. (1997). *Sybase SQL Server 11 Unleashed.* Sams Publishing.

[Rishe, 1992] Rishe, N. (1992). *Database Design: The Semantic Modeling Approach.* McGraw Hill.

[Rishe, 1996] Rishe, N. (1996). A file structure for semantic databases. *Information Systems*, 16(4):375–385.

[SDB, 1995] SDB (1995). *Semantic Binary Database C++ Interface Version 3.* High Performance Database Research Center, School of Computer Science, Florida International University.

[Urman, 1996] Urman, S. (1996). *Oracle PL/SQL programming.* Oracle Press, Osborne McGraw-Hill.