# REGULAR PAPERS

- **Dynamic Programming in Datalog with Aggregates**  
  S. Greco  
  [265]

- **Techniques for Increasing the Stream Capacity of A High-Performance Multimedia Server**  
  D. Jadav, A.N. Choudhary, and P.B. Berra  
  [284]

- **Resource Scheduling In A High-Performance Multimedia Server**  
  H.H. Pang, B. Jose, and M.S. Krishnan  
  [303]

- **Join Index Hierarchy: An Indexing Structure for Efficient Navigation in Object-Oriented Databases**  
  J. Han, Z. Xie, and Y. Fu  
  [321]

- **A Hybrid Estimator for Selectivity Estimation**  
  Y. Ling, W. Sun, N.D. Rishe, and X. Xiang  
  [338]

# CORRESPONDENCE

- **Proof of the Correctness of EMYCIN Sequential Propagation Under Conditional Independence Assumptions**  
  X. Luo and C. Zhang  
  [355]

**1998 TKDE Reviewers List**  
[360]

---

Dr. Naphtali Rishe  
Florida International University  
School of Computer Science  
Southwest 8th St. and 107th Avenue  
University Park  
Miami, FL 33199
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.

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
reges@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 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 Lochosky, 1989; Atkinson et al., 1989] - by their very definition - already deal with proceduralty. 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; Javorksi, 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 these 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 interject 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 "base 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].
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.

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 C++ example (Figure 1), such as OpenDatabase() and NewRelation() now appear as methods of class Proc.

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_get0bjects(O);
```

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 Solaris and Windows NT.
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 bytecode, 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 "birthYear", a constructor, and a simple method for age calculation. This class is quite similar to the Person category of Figures 4 and 5. 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.

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;

For clarity, we have omitted the Java housekeeping and exception handling code.
public static void main(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
    System.out.println(DB.transactionEnd());
    DB.transactionEnd();
    // 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
    System.out.println(DB.transactionEnd());
    DB.transactionEnd();
}

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 byte-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 built-in 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
Figure 9. Executing Java Class Code

Figure 10. Executing Java Class Code (continued)

Figure 11. Java Person Class

References


