22

98-MJ

compsac 98

The Twenty-Second Annual International Computer Software & Applications Conference

August 19–21, 1998

Vienna, Austria

Proceedings

The Twenty-Second Annual International Computer Software & Applications Conference

(Compsac '98)

August 19-21, 1998 Vienna, Austria

Sponsored by
IEEE Computer Society

IEEE COMPUTER SOCIETY

Los Alamitos, California

Washington • Brussels • Tokyo

A Modular Java API for Object-Oriented Databases

Raimund K. Ege, Yaman Battikhi, Philippe Pardo, Jinny Uppal, Naphtali Rishe
High Performance Database Research Center *
School of Computer Science, Florida International University
University Park, Miami, FL 33199
ege@cs.fiu.edu

Abstract

The object-oriented programming language Java is an ideal companion to an object-oriented database system. This paper describes our approach to provide a seamless application programmer interface. It is based on a modular architecture with components for database engines, a communications protocol and a JAVA API faciltator. The open architecture is flexible, scalable and distributed in nature.

1. Introduction

Object-oriented software development is considered standard practice. Even database systems [1] are becoming object-oriented in the mainstream[1, 4]. With that we need modern application programmer interfaces (APIs) that deal not in terms of functions but in terms of *objects*. In this paper we will describe an API from the Java[5] object-oriented programming language to object-oriented databases. We will actually describe a modular architecture that allows for a distributed layout of object-oriented access to a variety of database systems.

Our architecture has the following components (see Figure 1):

1. a Java front-end, suitable to be used in a Java application or an Java applet [2]. The Java front-end follows the ODMG object-oriented database system standard (ODMG 2.0) [3];

2. a substrate, or facilitator, that enables ODMG-like orthogonal persistence;

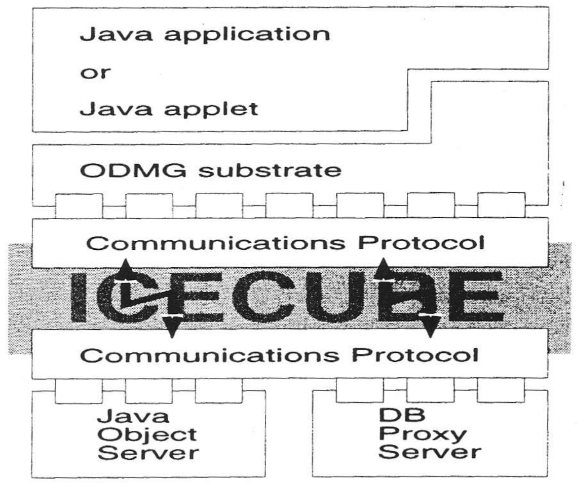

Figure 1. Modular Architecture

3. a communications protocol called Icecube, which enables objects to travel in a network of distributed clients and servers;

4. data base servers: either directly implemented database core engines, or proxy servers that interact with other database systems.

The paper is organized as follows: the next section describes our approach to provide transparent persistence to Java objects; then we will describe the Icecube communications protocol; then we will describe some potential database servers: (1) a core engine for an object-oriented database written in Java; and (2) a proxy server to the semantic binary database system [7]. The paper concludes with the status of our implementation and our future plans.

2. The Java API

Objects should be the unit of interchange between an object-oriented database and an object-oriented program-

*This research was supported in part by NASA (under grants NAGW-4080, NAG5-5095, and NRA-97-MTPE-05), NSF (CDA-9711582, IRI-9409661, and HRD-9707076), ARO (DAAH04-96-1-0049 and DAAH04-96-1-0278), DoI (CA-5280-4-9044), NATO (HTECH.LG 931449), AFRL (F30602-98-C-0037), and the State of Florida.

[1] Of course, we are aware that object-oriented database systems have been commercially available since the mid 80s.

[2] an applet is a program that runs within a Internet browser

ming language. Conceptually, an object-oriented database is a set of objects governed by a set of classes: so is an object-oriented program. The object-oriented program must be allowed to create objects in the database, retrieve objects from there and update them. The unit of intercourse has to be *object* !

The goal of our research was to facilitate the object-oriented view as much as possible: we want to allow a programmer to create and manipulate objects without being burdened with the tasks of database access and update. Following the standard outlined by the Object Database Management Group (ODMG, www.odmg.org) [3] we provide the concept of persistence to programming language objects. We support the concept of *orthogonality*: any Java object is allowed to persist, i.e. be stored in the database.

Our approach to extending Java with persistence capability is similar to PJava [2] and ObjectStore PSE [6], however, in our research we also focus on the architectural aspects of a distributed application programmer interface.

The issues involved in allowing Java objects to be persistent are (1) the selection/marking of classes that have persistent instances; (2) the creation of persistent objects; (3) the retrieval of objects from the database; (4) navigation from object to object; (5) updating objects; and also (6) some basic database housekeeping capabilities.

A Java class can be made persistent-capable, i.e. we allow that some of its instances reside in the object-oriented database, by defining it as a subclass of our `PObject` superclass. Any class in Java is already a subclass of Java class `Object`: we just extend that concept and require a `PObject` superclass. We can now say that any persistent object handled by our Java API is an instance of our `PObject` class. This class handles the basic correlation between the physical database object and its local Java counter part. The programmer, of course, needs not be aware of this.

Java objects can be created from Java classes using the regular new operator. Instances of a class that is a subclass of `PObject` allow the operation `persist()` which will declare an object to reside in the database world.

Objects that reside in the database are necessarily already persistent. We provide a naming facility: objects can be explicitly named. The name can be used to retrieve objects from the database.

Once a persistent object has been retrieved from the database, other objects are reached by following its relationships. Objects are automatically fetched as needed from the database. This concept is called *persistence by reachability*.

Updating objects is totally transparent to the programmer. The programmer invokes member functions and goes about the programming business as usual. Member functions can update member fields of persistent objects: that in effect is an update of a database object. The programmer does not have to insert any code to explicitly affect the update to the database. Of course, the final success of an update is dependent on the final success of the transaction.

```
import JavaOODBApi.*;

public class Person extends PObject {
  String name;
  int age;
  Person children[];
  public Person(String n,int a,Person c[]) {
     name = n; age = a;
     children = c;
  }
  public String getName() {...}
  public void setName(String n) {...}
  public int getAge() {...}
  public void setAge(int age) {...}
  public Person[] getChildren() {
      return children;
  }
  public void setChildren(Person c[]){
    children = c;
  }
}
```

Figure 2. Java API Person Class

```
Database db;
db = Database.open("demo");
Transaction tr = new Transaction();
// create objects
Person sophia = new Person("Sophia",3,null);
Person alex= new Person("Alexander",3,null);
Person children[] = { sophia, alex };
Person ray =
        new Person("Raimund", 38, children);
// make objects persistent
ray.persist();
// Create a named database root
db.bind(ray, "Raimund Ege");
tr.commit();
db.close();
```

Figure 3. Creating Person Objects

House keeping facilities are provided such as to connect to a specific named database server. Of course, handling of transactions can be specified and is passed on to the specific database server that is currently connected.

Figure 2 shows an example of a Java class `Person` that illustrates how little database detail is needed. The only database specific detail is the mention of superclass `PObject`. Class `Person` defines three attributes, one constructor, and six simple methods. Figure 3 shows a simple program that creates objects in the database. it first opens a "demo" database and starts a new transaction, which is basic database housekeeping. The program then creates three new `Person` objects. Database detail here is that after the objects have been created, object `ray` is made persistent with the operation `persist()`. The other objects automatically become persistent, since they are reachable from object `ray`. Object `ray` is then given a name, which can be used in subsequent object retrievals. Figure 4 shows a simple update program: after opening the same database and creating a new transaction, we first retrieve a named database object using the `lookup()` method. The rest of the program is straight Java code. The call `ray.getChildren()` accesses 2 dependent objects, in effect it retrieves 2 persistent database objects. The call `ray.setChildren(new)` updates a persistent object. Again, the programmer needs not be aware of this database access. The program concludes by committing the transaction and closing the database.

```
Database db;
db = Database.open("demo");
Transaction tr = new Transaction();
// retrieve named object
Person ray=(Person)db.lookup("Raimund Ege");
// retrieve dependent objects
Person old[] = ray.getChildren();
Person lucas=new Person("Lucas",0null);
Person more[]={old[0],old[1],lucas};
// update persistent object
ray.setChildren(more);
tr.commit();
db.close();
```

Figure 4. Updating Person Objects

The design of our Java API follows the ODMG standard closely. The Java code is compiled and matched to the ODMG substrate. In the ODMG substrate (see Figure 1) translates object creation, access and update into command of our Icecube communications language. The next section will discuss its design and implementation.

3. Communication Language

This section introduces the communication language that links the Java API to a database engine. Our architecture is not dependent on a specific database engine, rather it explicitly allows to connect to a variety of them. The communication language uses standard TCPIP sockets where client access is from the API side and the server is the database engine.

The API client access is logically done via a set of commands that allow a Java application program to access the services of the server database system. Java persistent object access is translated (in the ODMG substrate, see Figure 1) into Icecube commands. the commands travel on the socket link to the database server. The database server receives the command through its engine interface (EI). This EI is implemented as a set of switch statements that make the function calls associated with the method invocation information (Action Code) of an incoming message. The database engine delivers its results through its EI. The EI packages the results into a message and sends it out through the socket.

3.1. Format of the Communication Language

The communication language is called IceCube. The idea here is that the sending side (the condenser) will condense the message into an IceCube that can be easily melted by the receiving side (the melter). Our IceCube resembles the Java bytecodes of a .classfile. The cube is a bytestream. It contains a magic number, version number, the DataCube, and an ActionCube.

In detail, an IceCube has these entries: magic number of length 4 bytes; version number of 1 byte; overall length of the DataCube given in 2 bytes; a first DataCube structure: a typical structure has a type of 1 byte, a length field of 2 bytes, followed byte the byte values of the data; then maybe more DataCube structures; and finally an ActionCube structure: which identifies the active object with 2 bytes, an action code of 2 bytes, and then the arguments to the action with 2 bytes each.

The magic number identifies the data sent on the socket as an IceCube. The version number indicates changes in the scheme so a melter can take appropriate actions.

Entries of the ActionCube , except for the action code, are references to structures in the DataCube. The ActionCube has three main entries: the active object, action code and the argument list. A null active object (denoted as all "0") has no representative structure in the DataCube. The DataCube is an array of structures needed to interpret (melt) the ActionCube. A typical structure is composed of a type, a length, and a value. The first entry of the DataCube specifies the length of the array.

1	String	2	Integer	3	Float		
4	Double	5	.classFile	6	boolean		
7	class	8	void	9	object	10	array

Figure 5. IceCube Type Entries

Figure 5 a table of type entries and their associated types. Notice that an array is specified as the following special structure: type 10 followed by array length and the list of the contained structures. The other types use the typical structure defined above. We need to build a table of action codes and their associated functions. The first three bits of an action code refer to the category of request that the action code represents: 000 is for schema requests; 001 is for data requests; 010 is for queries; 011 is for exceptions; and 111 is for connection requests.

Figure 6 shows the table of action codes and their associated functionalities.

11100000	Ok
11100001	Error
00000000	createCategory(.classFile)
00000100	categoryDelete()
00000101	categoryRead()
00100000	createObject()
00100011	objectDelete()
00100100	objectRead()
00100101	objectUpdate(data)
01000000	setObjectName(globalName)
01000001	getObjectID(globalName)
01000010	categoryInstances()
01000011	categoryInstances(conditionList)
01000100	createDataBase(dataBasename)
01000101	closeDataBase()
01000110	openDataBase()
01100000	terminateConnectio()

Figure 6. Action Codes and Functions

The ODMG substrate passes an IceCube to the engine interface. The active object in the ActionCube part of the IceCube refers to the object id of the object on which a method is to be invoked. The engine will return an IceCube where the active object refers to the object id returned by the method invoked; the action code specifies the result status of the transaction (error, ok,...); the argument list contains a set of records returned as a result of the method invocation.

For example, assume we have an object id 010101 that refers to the class "Person". A user creates a new Person object P and wishes to store it into the database. Person p = new Person(); status = p.persist(); Here is the function we need to call: Person.createObject(). We need to condense the function call into this IceCube:

11 10 11 14.1 0 1 7 0 6 48 49 48 49 48 49 0 1 0 64 0 0

On the receiving side, we now need to melt the incoming IceCube: We read magic number 11 10 11 14, and version number 1. We know there is one structure in the DataCube because of the array length 01. The type of the structure is a class: 7. The length is 06 and the value is 48 49 48 49 48 49 which represents 010101. Now, the ActionCube has its active object referred to structure #01 previously described as class with object id 010101. The action code 0 64 refers to createObject in the action codes table. We have 00 argument. The receiving side successfully interprets the IceCube as: 010101 064 00 which is Person.createObject(). This scheme is very appealing because of its scalability and its flexibility. New types, new structures, and new action codes can be easily accommodated. The melter can make use of the information carried by a structure.

4. Database Engine Components

Our architecture allows for multiple servers to connect to an Java API via our IceCube communications link. In the following we will describe the 2 servers that we have currently running in our prototype implementation. Both servers are based on the semantic binary database model [7]: the first is implemented using Java; the second is represented with its C++ API, which maps itself into 2 available servers, one written again in C++, the other simulated on top of a relational database server.

4.1. Java Database Engine

The Java database engine is designed to follow the physical storage model of the Semantic binary database system [8]. A semantic binary database is defined by categories and their relations. A category defines a real world entity such as Person, Student, Course etc.. Categories can have relations, which in object-oriented terminology would cover both attributes and associations. Suppose the category Person has a relation "name" of type (range) "String". In semantic database terminology the category Person has a binary relation with the category "String" and this relation goes by the name "name".

All objects in the database, categories, relations and their instances have unique identifiers assigned to them. The only objects that do not have IDs are the primitive datatypes (int, float etc.)

All information in the database is represented as one of the two binary facts "aC" and "xRy" where a,x and y are IDs for objects (including primitive datatypes), C is the ID for a category and R is the ID for the relation that associates x with y.

"aC" indicates that the object is of type C, i.e. it belongs to the category whose ID is C. "xRy" indicates that object x has the relation R with the object y. The semantic database is a set of such facts.

Figure 7 shows an example of a database schema with categories Person, Student and Course.

Suppose now that we have two Course instances (with identifiers C1 and C2) and one Student instance (S1) in our database. The data stored (in the format of "aC" and "xRy") is:

```
C1 Course (aC)
C2 Course (aC)
S1 Student (aC)
S1 Person (aC)  (Note: inheritance)
C1 name "Introduction to AI" (xRy)
C2 name "C Programming" (xRy)
```

```
Category Person
Relation#1, Name: name  , Range: String (1:1)
Relation#2, Name: age   , Range: Integer(1:1)
Category Student SubCategoryOf Person
Relation#1, Name: course, Range:Course (1:m)
Category Course
Relation#1,  Name:name   , Range:String(1:1)
```

Figure 7. Semantic Binary Database Schema

```
S1 name "John" (xRy)
S1 age 34 (xRy)
S1 course C1 (xRy)
S1 course C2 (xRy)
```

Along with the above binary facts we also store inverses. The inverse of a fact of type "aC" is "Ca". As for facts of type "xRy", there are two possibilities, "y" could be an instance of another category (association between two categories) or a primitive datatype (e.g. int, String etc.). The former fact is denoted by "xRy" and the latter by "xRv". In case of a fact of type "xRv", the inverse is stored as "Rxv" and for facts of type "xRy", the inverse is "yRx". Storing inverses introduces redundancy in the database, however the payoff in fast retrieval of data compensates for the redundancy in the database (see [8] for more details).

The data structure used to store the binary facts is a B+Tree. All facts are stored at the leaves. Since a B+Tree stores keys in the order of its fields, the binary facts are stored in lexicographical order. That means all facts describing a particular object are stored in serial order, since the prefix ("a" in "aC" or "x" in "xRy") is the same. A group of facts are stored in one disk block and one disk access loads the entire block into memory. A block can therefore then be read to retrieve a large number of facts. The index block of the B+Tree is stored in memory to facilitate faster search. The basic B+Tree operations of insert, find and delete allow the atomic queries to be satisfied.

All queries can be broken down into the following atomic queries:

aC	verify the fact aC
xRy	verify fact xRy
a?	find categories the object a belongs to
?C	find all instances of category C
aR?	get all y such that aRy
?Ra	given object a find all y such that yRa
a?+a??+??a	all information on the object a
?Rv	find all x such that xRv
?R[v1,v2]	find all xwhere xRv1 ¡= xRv ¡= xRv1

4.1.1 Engine Interface(EI)

The engine interface (EI) is a set of classes that provide the API access to the database. This component lets the API store and manipulate data and perform other database tasks.

The engine has as one of its components the "Melter" which listens for icecubes from the communications link. When an IceCube is received over the network, the Melter breaks down the IceCube in to its various components and extracts the command and the data from the IceCube. The engine interface maintains a table of codes (the class and/or method number ,see Fig 6) and the corresponding EI methods to be invoked. The prototype of the methods is stored in the table, so it knows what parameters are to be expected in the input stream. The Melter uses this table as reference to look up the method that has to be executed.

A "DataBase" class is the "do-all" class which serves as the main interface for most of the tasks that can be performed on the database. Apart from allowing a user to create, open and close a database, it provides methods to create and delete objects and the facts that describe them. The DataBase class also allows the user to define the schema (explained in greater detail in the next section). It has methods to execute simple atomic queries on the database. Complex queries are broken down to a series of atomic queries before they passed onto the DataBase methods. A DataBase object keeps track of all the Objects in the database. The DataBase class also allows objects to be named. So an object may be assigned a unique name (String) which is internally stored in the database system. The object can then be referenced anytime in the future by this name. The name can be set or unset. Moreover, given a name the corresponding object can be retrieved from the database.

There are two types of data maintained in the database: data (actual instances of categories) and meta-data(data about categories and relations). For the meta-data we introduce a category called "Schema" which maintains all the data defining the schema of the database in question. An object (instance) of the Schema category has as its members all the categories and the relations in the database. Every time a new category is added, the schema object changes. We also have categories called "Category" and "Relation", which as the names suggest describe a category and relation respectively. An instance of category has as its members: the name of the category, the super-categories and the relations of that category. The "Relation" category has as members the name of the relation, the domain (which is a category) , the range (could be a category , primitive datatype or user-defined datatype) all of which are stored in the database.

4.2. Other Engines and Issues

Our second database engine server is written in C++, since our existing semantic binary database engine has a C++ API.

The C++ server consists of three major parts. The first part is the "Melter", which deals with communication with

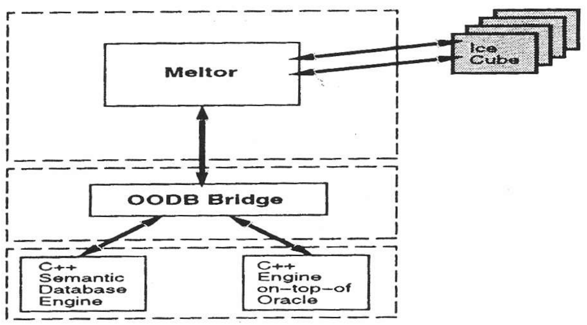

Figure 8. C++ Engine Server Architecture

various clients using the IceCube communication language. When an Icecube is received from a client, the Melter breaks it down to its smaller components, and depending on the Icecube action code the requests will be passed to the second part which is called "OODB Bridge". The Melter is also capable of constructing an acknowledgment Icecube depending on the client requested action code. Figure 8 shows the organization. The "OODB Bridge" is where Java API requests are translated into requests to the semantic binary database system C++ API [9]. This API allows access to a pure C++ based Semantic Database Engine or to our C++ Semantic Database engine simulated on top of Oracle relational Database.

To represent a Java class in the semantic database meta schema, the client will send an Icecube containing the compiled class file of that class with createCategory action code. After melting this IceCube in the server side the Melter will extract the binary class file and pass it to "OODB Bridge" with a request to create a new semantic database category. In the "OODB Bridge" we have developed a Java class file interpreter, which is capable of retrieving all information about a Java class from its compiled binary format including its superclass , its private and public data fields and its private and public methods.

A semantic database category is created for each Java class. To store an instance of a class in a semantic database the client will send an Icecube containing a Java serialized object together with the appropriate action code. After melting this IceCube in the server side the Melter will extract the serialized object and pass it to "OODB Bridge" with a request to store this object in the semantic database. The OODB Bridge first will check to see if this object has a valid category in the semantic database and then will use a method to break down the serialized object and extract its data fields and create an semantic database instance of the category representing that object and store it as a binary fact.

There are two methods available to retrieve the serialized object. The first is to invoke a Java virtual machine within the C++ code and to try to create a real Java object and then to extract its data one by one. The second method is to develop a C++ Java serialized object handler which can retrieve the data fields from the serialized object as well as constructing a serialized object from any semantic category object.

We have also developed tools that use the semantic database API to manipulate data in an Oracle relational database. The semantic database concept has been simulated on Oracle. The physical layout of the Oracle relational database is implemented by creating one big table with one column. Each row of this column will contain a semantic binary fact such as "ac", "aRy" and "aRv" or its inverse binary relation fact. All facts are stored in a compressed format. In addition, the table is fully indexed (see also Section 4.1). With "Embedded SQL" inside C++ we are then able to support our C++ semantic API on top of Oracle. This semantic API then will be used the same way the pure Semantic database engine API is used to store serialized objects.

5. Conclusion

In this paper we presented a distributed and scalable architecture for a Java application programmer interface. Our prototype implementation was done on a network of Windows NT and Solaris based workstations. Our next goal is to use this API to implement a Java-based database administration tool.

References

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

[2] M. Atkinson, L. Daynes, M. Jordan, T. Printezis, and S. Spence. An orthogonally persistent java. *ACM SIGMOD Record*, 25(4):68–75, 1996.

[3] R. Catell, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, and D. Wade, editors. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann, 1997.

[4] R. Ege. Database support for object-oriented simulation. *International Journal of Systems Engineering*, 1994.

[5] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison Wesley, 1996.

[6] Object Design, Inc., http://www.odi.com/content/products/-pse/doc_120/doc/apiug/index.htm. *ObjectStore PSE and PSE Pro for Java User Guide*, 1998.

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

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

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