

# Java Universal Binding: Storing Java Objects in Relational and Object-Oriented Databases

Florian Xhumari<sup>†\*</sup>

Cassio Souza dos Santos<sup>\*</sup>

Marcin Skubiszewski<sup>†\*</sup>

August 29, 1997

<sup>†</sup>INRIA, Rocquencourt  
78153 Le Chesnay, France  
*FirstName.LastName@inria.fr*

<sup>\*</sup>O<sub>2</sub> Technology  
7, rue du Parc de Clagny  
78000 Versailles, France  
*http://www.o2tech.fr*

## Abstract

We introduce JUB (*Java Universal Binding*), a software tool that stores Java objects in relational and object-oriented databases. JUB supports the object-oriented DBMS O<sub>2</sub>, the relational DBMS Oracle and Sybase, and all the relational databases which can be accessed via JDBC.

In the context of O<sub>2</sub>, Java objects stored in the database are first-class database objects: they can be accessed by all the clients of O<sub>2</sub> (O<sub>2</sub> supports application programs written in C, C++, Smalltalk, and O<sub>2</sub>C).

We describe JUB from the application programmer's point of view. We discuss the architecture of JUB, and the way in which Java classes and objects are translated into O<sub>2</sub> types and objects. We describe the current status and the performance of the product.

## 1 Introduction

Many different approaches have been proposed to allow Java programmers and applications to take profit from database technology. These approaches can be divided into two main groups: those proposing an extension to the Java language or virtual machine, and those taking the language as it is and defining a persistence service layer on top of the Java language and of an existing database system.

The first group aims at defining a persistent version of Java that requires a nonstandard compiler and/or virtual machine. For example, the systems described in [2, 1, 6] belong to this group.

The second group can be further divided, according to the level of integration between the database system and Java, into database drivers (JDBC) and language bindings (ODMG).

JDBC drivers [7] provide access to existing databases through an API that reflects the underlying database model. Applications must map the native structures of the database (in the relational model, rows and columns) into corresponding Java objects and attributes. Many JDBC drivers are currently available or under development. A ODBC/JDBC bridge is also available.

Language bindings provide transparency to persistence in that the mapping between Java objects and the underlying database structures is performed automatically by the runtime system. Applications manipulate persistent objects as if they were ordinary (transient) Java objects. Persistence is requested either explicitly or indirectly, through the attachment to a persistent root.

The ODMG Java binding is defined in version 2.0 of the standard. Most object database vendors have announced Java bindings to their systems.

JRB (*Java Relational Binding*) [9] is a Java binding to relational databases. It was initially defined on top of Oracle and Sybase databases, and was later extended to run on top of a JDBC driver. JRB provides an API that allows applications to manage database entities (bases, transactions, queries) and to store and retrieve Java objects into/from the underlying database.

The work described in this paper is built on top of JRB. Here, we do not describe the way in which Java objects are stored in a relational database (this issue has been treated in [9], where a complete description of JRB is given). Instead, we describe a general architecture featuring a common runtime running on top of different DBMS. This architecture is called the Java Universal Binding (JUB). A common API running on top of JUB allows applications to access relational and  $O_2$  databases undistinguishably and in a transparent way.

$O_2$  Java is a specific component of the JUB architecture providing access to  $O_2$  DBMS [3]. It is analogous to JRB, but this time the target database system is the  $O_2$  DBMS.

JUB uses the same algorithm as JRB to store Java objects in a relational database. No other aspects of JUB are inherited from JRB. Most notably, the structure of the product is novel: a well-defined interface has been introduced between the higher-level part of the product, which is independent on the DBMS being used, and the lower-level part, which is DBMS-specific.

The paper is organized as follows. Section 2 describes JUB as seen from the user's point of view. Section 3 describes the structure of the product. Section 4 describes the way in which we translate Java classes and objects into the corresponding  $O_2$  types and objects. Section 5 describes the current status of the project. Performance is discussed in Section 6.

## 2 Using JUB

This section introduces the key properties of JUB, as seen from the application programmer's point of view.

JUB is a Java database binding, *i.e.* a tool that makes it possible to store Java objects in a database. With JUB, the application programmer is aware of the fact that there is a database involved and that some objects are persistent and are stored in the database, whereas others are not. He/she invokes database synchronization primitives like `transaction()`, `commit()` or `abort()`, and knows that these invocations affect persistent objects. The user may explicitly take database locks (although this is usually unnecessary, because JUB automatically takes the appropriate locks whenever an object is read or written).

Our approach is different from the one used in Persistent Java [2], where every attempt is made to give to the system the appearance of an ordinary (non-persistent) Java runtime system.

### 2.1 Persistence-capable types

When JUB is used, the application programmer has access to two kinds of Java objects: ordinary or *transient* objects, which behave as if JUB did not exist, and *persistent* objects, which are managed by JUB, and are stored in a database. In order to be persistent, an object must belong to a *persistence-capable* type.

A class or an interface is persistence-capable iff it satisfies two conditions. First, it must implement or extend the interface `PersistentObject` from the package associated with JUB (namely, package `jub.api`). This interface contains all the methods necessary to manage persistence-related properties of objects. Second, the class must have been *imported* by JUB. The import operation creates in the underlying database the data structures necessary for storing objects of a given class: in a relational database, appropriate relations are created [9]; in  $O_2$ , a type is created that corresponds to the Java class (see Section 4).

An array type is persistence-capable iff an array of this type can be referenced to by an attribute of a persistence-capable class and the type of the elements of the array is persistence-capable. The import operation detects these conditions and creates the necessary structures in the database.

## 2.2 Making objects persistent

Two models have been proposed for choosing which objects should be made persistent: explicit persistence and persistence by attachment.

With explicit persistence, the application programmer explicitly requests objects to be added to or deleted from the database, *i.e.* to be made persistent or to be made transient again.

With persistence by attachment, a fixed set of objects is made explicitly persistent. These objects are stored in the database and are directly accessible to the application programmer. They are called *roots of persistence*. For all other objects, the following rule is applied recursively: if a persistent object *a* contains a pointer to an object *b*, then *b* is also persistent. This rule can be equivalently expressed in a non-recursive form: an object is persistent if it is *reachable*, *i.e.* if and application program can reach it by starting from a root of persistence, and by following pointers from one object to another.

With persistence by attachment, objects that are persistent, but are no longer reachable, are deleted from the database by a garbage collector.

JUB implements both kinds of persistence. In both cases, the static members of persistence-capable classes can be accessed directly (*i.e.* without the need to first obtain a pointer to the object) by application programs. Static members are made persistent at import time, except if the user requests otherwise.

The user can request JUB to maintain *class extents*—collections containing all the objects of a given class. When class extents exist, they are directly accessible to application programs.

With persistence by attachment, the static members and the class extents are roots of persistence.

## 2.3 Using a persistent object

The usage of persistent objects involves three specific issues. First, the user program must notify JUB of reads and writes performed on persistent objects. Second, under certain circumstances, persistent Java objects are *detached* from the database: changes made to detached objects will not be written into the database. And finally, the application program may label certain fields as *transient*. Such fields will not be stored in the database, although they exist in the in-memory version of the object.

### 2.3.1 Access notifications

Before accessing a persistent object, a Java program must inform JUB of its intention. This is done by invoking the method `access` for the object in question. This invocation causes the object to be actually loaded in the address space of the program. Until then, it may be the case that the real object is only stored in the database, and what appears to be the object in the address space of the Java program is in reality a *shadow object*: a placeholder object that has the appropriate type, but whose contents are meaningless.

The method `access()` is defined for all persistence-capable types.<sup>1</sup> `access` has no effect when invoked for a non-shadow object, either persistent or transient. Invoking `access` many times in a row for the same object is equivalent to invoking it once. Therefore, there is no harm in calling `access` too many times, or in calling it for a transient object. This simplifies the use of `access`: application code can invoke the method before every access to an object that belongs to a persistence-capable class, without knowing

---

<sup>1</sup>For array types, it is impossible to define methods. Therefore, if `t` is an array, `t.access()` is not defined; instead, we use the static method `access` in class `jub.Database`, like this: `Database.access(t)`. The same remark holds for the method `markModify`, mentioned below.

whether the call is redundant or whether the object is actually persistent. Application code can therefore be instrumented so as to call `access` in a systematic way, before every access to a persistence-capable object. The instrumentation relieves the programmer from the necessity to add calls to `access` by hand. It is done automatically, by a postprocessor of Java bytecode.

### 2.3.2 Write notifications

When a persistent object is modified by a Java program, this fact must be notified to JUB, so that when the current transaction commits, JUB will propagate the modification to the database. The notification takes the form of a call to the method `markModify`.

`markModify` is only intended to be used with persistent objects. `markModify` is similar to `access` in that it can harmlessly be invoked outside of its intended scope of use (namely, for transient objects) or be invoked many times, instead of just once. It is therefore possible to instrument application code so that every modification of a persistence-capable object is followed by a call to `markModify`. The instrumentation is done automatically, by the same postprocessor which adds calls to `access`.

### 2.3.3 Invalidating persistent objects

JUB can *invalidate* a persistent object, *i.e.* put the object in a state in which it can no longer be used. All objects are invalidated when the current transaction commits. This is necessary, because at commit time all locks are released, and therefore the state of the persistent objects, as represented in the address space of the Java program, is no longer guaranteed to correctly represent the real contents of the objects.

Additionally, the application program can request the invalidation of individual objects. This is useful (and sometimes necessary) in order to release the resources used by objects that have been accessed at some point, but are unlikely to be accessed again.

The application program must not use invalidated objects. It is recommended that the program destroys all the references to such objects, so that the Java garbage collector can destroy them.

### 2.3.4 Transient fields

Fields in persistence-capable classes can be labeled as `transient`. Such fields are never stored in the database. Therefore, when a persistent object is brought from the database to the address space of a Java program, transient fields need to be properly initialized. This can be accomplished by the method `activate()`: if this method is defined for the object, it will be invoked every time the object is brought to the address space. The rôle of this method is limited to initialization of transient fields of the object. In particular, it is not allowed to modify any persistent attribute or otherwise manipulate a persistent object. The system enforces this rule.

## 2.4 Storing Java bytecode in the database

It is possible to store Java bytecode in the database. The bytecode of the classes stored in this way can be loaded into the execution environment by means of the class `DatabaseClassLoader`. This class extends the class `ClassLoader` and provides the mechanism to load the bytecode, pass it to the Java runtime making it possible for the program to manipulate objects belonging to the loaded class. The user may explicitly use the class `DatabaseClassLoader` to load a particular class. But the most interesting use of stored bytecode is when the database contains objects of a class which is not available to the program from the `CLASSPATH`. If the program accesses such an object, the JUB loads the implementation of the class from the database, allowing the object to be properly used.

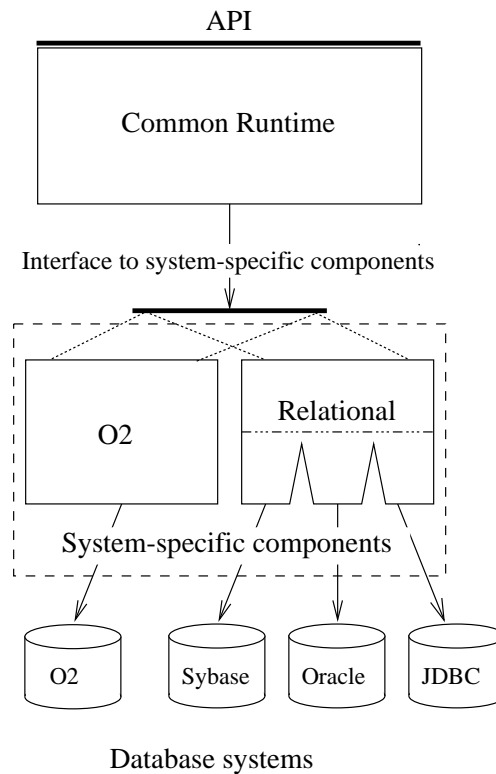


Figure 1: The runtime system of JUB.

Let's give an example of the utility of this feature. A class `C` which implements the interface `Runnable` is stored in the database and an object belonging to the class is created. The `Runnable` interface defines a method `run()` and the implementation of this method in `C` performs some action. An external program needs to know only the `Runnable` interface; it accesses the stored object and calls its `run()` method and the proper action is performed.

### 3 The architecture of the runtime system

The runtime system of JUB is divided into a *common runtime* and *system-specific components* (Figure 1). The common runtime is accessed by application programs through an API (*application programmer's interface*). There are two system-specific components: one for relational databases, and one for `O2`. The component specific to relational databases contains three sub-components, which access, respectively, Oracle, Sybase, and JDBC. Access to Oracle and to Sybase is done via their respective native SQL interfaces (incidentally, one can also access Oracle and Sybase through a JDBC interface, but this is less efficient).

Both system-specific components have the same interface, through which they are used by the common runtime. They allow the common runtime to connect to a database, to manage transactions (perform operations like `commit()` or `abort()`), and to create, read and modify persistent objects in the database. All these operations are performed without the common runtime knowing whether the database is object-oriented or relational: the common runtime always views the database as a repository of objects.

The system is designed so that more system-specific components, accessible through the same interface,

can be added, and can be used by the common runtime.

The common runtime is based on two data structures: the *object table* and the *metadata classes*. Let us describe these structures briefly.

### 3.1 The object table

The object table is a transient data structure, internal to the common runtime. It contains information about the persistent objects that exist in the local address space. For every such object, it memorizes the object's address in the address space, its type, its state, and its *cache identifier*.

The cache identifier is used to identify the object when communicating with a system-specific component; from the common runtime's point of view, it plays the rôle of the object's address in the database. The cache identifiers of objects are computed by the system-specific component, in a way that allows for easy mapping between the cache identifier and the information necessary to find the object in the database.

The state of the object is either *shadow*, *accessed*, *modified*, or *new*. *Shadow* is the state of shadow objects, *i.e.* of objects that are present in the database and are referenced from within the local address space, but have not been properly copied into this space (see Section 2.3.1). *Accessed* is the state of objects that are present both in the database and in the local address space. *Modified* is the state of objects that are present both in the database and in the local address space, and for which `markModify()` has been called. *Modified* objects are written to the database at commit time. *New* is the state of the objects which have been made persistent by the local application program, during the current transaction, and have not yet been added to the database. *New* objects are added to the database at commit time.

### 3.2 Metadata classes

The metadata classes contain information about persistence-capable classes. There is one metadata class per persistence-capable class. Information stored about each persistence-capable class includes its name, the name of the parent class (if any), the names and types of the attributes of this class and those of the static attributes of the class.

Metadata classes are generated by the import tool. If the bytecode of a persistence-capable class is stored in the database, then the bytecode of the metadata class is stored there, too.

To use a metadata class, the common runtime creates an object belonging to this class (the object is transient), then accesses the object through the `ClassMetaData` interface. The methods of this interface return information about the persistence-capable class corresponding to the object in question. All metadata classes implement the interface `ClassMetaData`.

The common runtime contains a *metadata manager*—a module capable of finding (or, if necessary, for creating) metadata objects corresponding to any given class name. When the manager needs to create a metadata object whose class is not present in the execution environment, the class bytecode is loaded from the database, as described in Section 2.4.

## 4 The mapping of Java classes into O<sub>2</sub> types

The mapping from Java classes into O<sub>2</sub> data structures benefits from the similitudes between between the Java type system and the O<sub>2</sub> data model. Incompatibilities also exist, and we discuss the ways they are handled.

Table 1 shows in detail the mapping between Java types and O<sub>2</sub> types.

**Classes, interfaces and objects** Java classes and interfaces are translated by the import tool into O<sub>2</sub> classes (an O<sub>2</sub> class corresponding with a Java interface contains no attributes). When a class or an interface is imported, all the interfaces and classes from which it inherits (*i.e.* which it extends or implements) are also imported. The inheritance structure is entirely preserved by the translation process. This is possible due to the multiple inheritance mechanism of O<sub>2</sub>, whose semantics englobes that of Java.

When a subclass defines an attribute with the same name as an attribute in a superclass, O<sub>2</sub> considers that the attribute is overloaded, whereas Java considers it completely distinct from the superclass' attribute. To overcome this problem, in case of a conflict we create the attribute with a unique name and then rename it to the original name. This causes O<sub>2</sub> to consider the two attributes as different attributes. The runtime system keeps track of renamings so as to load the corresponding Java attributes accordingly.

**Scalar types** shorts and ints are translated into integers. floats and doubles are translated into reals. If there is an overflow in the process of this conversions, an exception is thrown.

Java 64-bit long values are mapped into pairs of 32-bit integer O<sub>2</sub> integer attributes. The runtime system performs the appropriate data conversion when long values are loaded from the database into long variables.

**String, Integer, Float,...** Objects belonging to `String` and to classes of simple types (like `Integer`, `Float`, etc) do not retain their identity in the database. In this respect, our system changes the semantics of Java language, but we considered that having these objects retain their identity would be too expensive in terms of performance. Furthermore, the values encapsulated in these objects are immutable, which limits the non-transparency to the only case of reference comparison.

Unicode characters in `Strings` are converted and stored in the database as ASCII. It is possible to specify in the configuration file that they are to be stored as Unicode. Unicode strings are stored always as UTF strings; the conversion is performed by Java JNI (Java Native Interface) runtime.

**Arrays** A Java arrays is mapped into an O<sub>2</sub> class encapsulating a list. When a new array is created in the database, the whole corresponding list is filled with nulls. When a list is read in as an array, the size of the list determines the size of the array to create.

Arrays in Java follow the same hierarchy structure as the types of their elements. This structure is closely followed by the corresponding O<sub>2</sub> classes which implement arrays. Such classes are of type list, and the model of O<sub>2</sub> allows inheritance between lists to follow the inheritance between their elements. From this point of view, the O<sub>2</sub> model matches exactly the Java model.

Arrays of Java long values are implemented using lists of 32-bit integers, considering two consecutive elements as a single 64-bit integer.

**Static variables** Static variables are mapped into O<sub>2</sub> names, thus becoming persistent roots of the database.

## 5 Current Status of the Implementation

An early version of JUB exists today and has been tested. This version has some restrictions. Only explicit persistence is implemented. Persistence by attachment is not available today, although the major difficulties connected to it have been solved: a garbage collector for O<sub>2</sub> exists [8], and persistence by attachment is available in all language bindings to O<sub>2</sub> except Java.

The application programmer must instrument Java code with calls to `access` and to `markModify` by hand. The postprocessor of Java bytecode, which will perform this task automatically, is under development.

<b>Java type</b>	<b>O<sub>2</sub> type</b>
<b>Builtin types and corresponding classes</b>	
boolean, Boolean	boolean
char, Character (16 bit, UNICODE)	char (8bit ASCII), or integer (16 bit, UNICODE)
byte, Byte (8 bit)	char (8 bit)
short, Short (16 bit)	integer (32 bit)
int, Integer (32 bit)	integer (32 bit)
long, Long (64 bit)	two attributes of type integer, whose names are suffixed by hi and lo
float, Float	real
double, Double	real
String (UNICODE)	string (UTF coding, as defined in Java JNI)
<b>Classes, interfaces, arrays</b>	
interface I { ... }	class I (no data)
interface J extends I { ... }	class J inherit I (no data)
class C { ... }	class C public type tuple(...)
class C extends S implements I { ... }	class C inherit S, I public type tuple(...)
<i>JavaType</i> []	class o2_list_ <i>O2Type</i> public type list( <i>O2Type</i> )
<i>JavaType</i> [], where class <i>JavaType</i> extends S implements I ...	class o2_list_ <i>O2Type</i> inherit o2_list_ <i>S</i> , o2_list_ <i>I</i> public type list( <i>O2Type</i> )
<i>JavaType</i> [][]	class o2_list_o2_list_ <i>O2Type</i> public type list(o2_list_ <i>O2Type</i> )
<b>Special cases</b>	
long[], Long[]	like int[], with two successive elements coding the low and high part of a long
byte[], Byte[]	class ByteArray type public bits
char[], Character[]	class ByteArray type public bits
java.lang.Vector	o2_list_Object, of type list(Object)
<b>Static variables</b>	
in <i>class</i> , static <i>JavaType att</i>	name o2_var_class : tuple ( <i>att</i> : <i>O2Type</i> ... )

Table 1: The mapping of Java types to O<sub>2</sub> types.

Collection classes in the style of those defined in the ODMG Java Binding [5] (sets, bags and lists) are being currently implemented and should be soon available in JUB. At short term, JUB API should evolve towards an ODMG compliant interface.

Today, it is possible to launch OQL or SQL queries in the underlying database. We are in the process of investigating the possibility of the execution of Java methods from an OQL query.

## 6 Performance measurements

We present preliminary performance measurements of our prototype using the O<sub>2</sub> system. We investigate the impact of the differences between JUB and the C++ binding to O<sub>2</sub>, as well as the scalability of JUB.

### 6.1 The benchmark

Our measurements are based on the OO1 benchmark [4]. We used OO1's database schema and two operations: traversal and creation. We did not include the lookup operation and the reverse traversal operation of OO1 benchmark because we felt that performing these operations would not add significant information to our measurements.

We implemented a simple benchmark because our goal is to measure the impact of the language binding and not the performance of the database system itself.

We use a database composed of objects called *parts*. Each part contains an identifier, a date, a part type and two coordinates:  $x$  and  $y$ . Each part points to three other parts. It also contains a list of parts from which it is referenced—the reverse links. For each reference to a part there is a type and a length. The type fields are strings of ten characters, the date is a 64-bit integer. The identifier, the coordinates and the length fields are 32-bit integers. A global hash table contains all parts, hashed by the part identifier.

Two database sizes were tested: one *small* database containing 20000 objects and one *large* database with 200000 objects. The fields of each object data occupy about 100 bytes and if we include the space overhead of other structures, this gives databases of about 4 Mb and 40 Mb respectively.

The database is constructed in two phases. First, the parts are created with links set to null. The identifiers of the parts are set in increasing order, from 1 to the total number of parts, while the other fields are set to random values. In the second phase, we link each part to three other parts; the reverse links are updated as appropriate. The parts to link are chosen so as to obtain some locality of reference: 90% of them are randomly selected among the 1% of the parts that are “closest”, and the remaining 10% are randomly selected from all parts. The parts are said to be close if their identifiers are numerically close. The global hash table is used to find a part, given its identifier.

Our first experiment is a traversal in the graph of parts. Starting from a random part, we visit all parts connected to it in a depth-first order, up to a depth of 7 levels. In total this makes 3280 parts, with possible duplicates. For each part visited we read the values of all attributes. This experiment is executed in read-only transactional mode.

The second experiment is similar to the first one, with the difference that for each part visited, the field  $x$  is incremented. The time measured includes the time to commit the changes to the database.

The third experiment inserts new parts into the database. A new part is created and is linked to other parts following the same rules used to construct the database. We create 100 parts with increasing part identifiers and report the time needed to create the parts and to commit the transaction.

We measure execution times of our tests in two different runs: a cold run, when there is no data cached in the database server or in the client, and a warm run, where cached data are present. Practically, we run the tests 15 times and consider the first run as a cold run and the last 7 times as warm runs.

## Traversal warm times

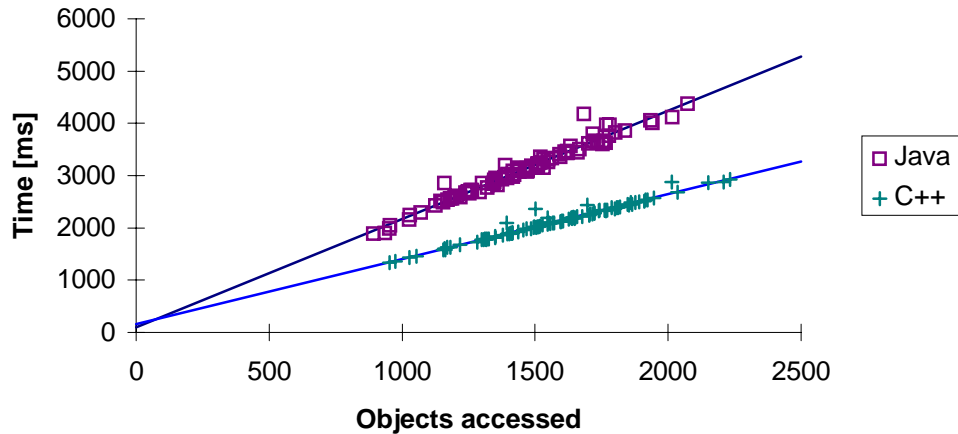


Figure 2: Traversal time depending on the number of distinct objects visited

### 6.2 System configuration

For our benchmark, we used a Sun UltraSparc 1 workstation with 128 Mb of random access memory, 192 Mb of swap space and two 4 Gb disks connected to two different fast-wide SCSI controllers. One disk contained only the database files and the other contained all the system files and database software. During the measures, the system was used exclusively by our benchmark.

Both the server and the client processes ran in the same machine. The server was configured with 4 Mb of cache and the client had 4Mb of cache too. We used O<sub>2</sub> version 5.0.2.A.4 and the Java Virtual Machine included in Sun's JDK 1.1F.

### 6.3 The results

Figures 2 and 3 show the results of the traversal and modification experiments in the small database, the warm run. *Objects accessed* is a count of distinct objects visited during the traversal.

We observe significant differences in the number of objects accessed in each run, as well as a difference in the time measured. The time is almost proportional to the number of distinct objects accessed. This suggests that the first access to an object is time-consuming while further accesses are much faster. This is an expected result. In the case of Java, the first visit to an object causes the object fields to be loaded from the database and shadow objects to be created for the three pointed-to objects. A subsequent visit only causes the system to query the object table about the state of the object. In C++ things are much the same: during the first visit the object itself is created and its fields are filled in; subsequent visits just test a bit in C++'s reference variable (`d_Ref`).

Given this behavior, we decided to present in our results the execution time divided by the number of distinct objects accessed. For the insert experiment, we divided the time by the number of objects inserted (100).

Table 2 shows the results we obtained in our experiments on the small database and table 3 shows the results for the large database.

### Modify warm times

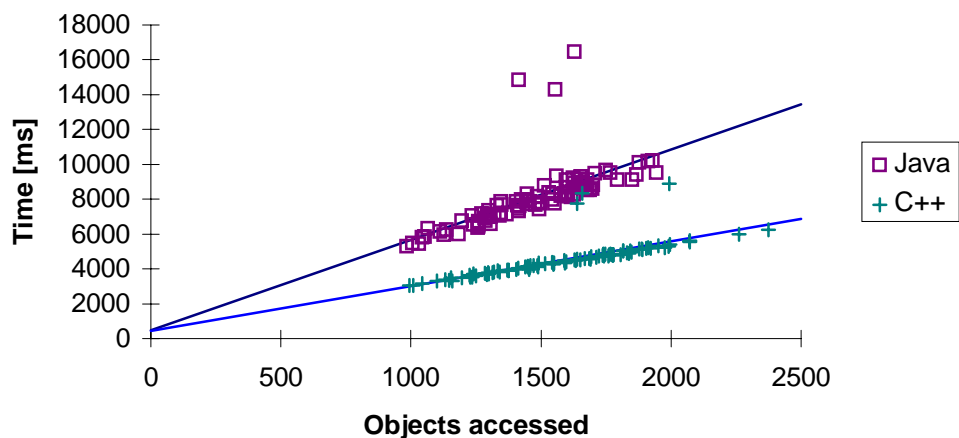


Figure 3: Modify time depending on the number of distinct objects visited

	Time		Percent slower
	Java	C++	
cold traverse	4.95	2.22	123%
warm traverse	2.14	1.35	59%
cold modify	7.13	3.78	88%
warm modify	5.51	2.86	93%
cold insert	23.39	18.05	30%
warm insert	16.81	12.49	35%

Table 2: Small database results

	Time		Percent slower
	Java	C++	
cold traverse	6.57	3.03	117%
warm traverse	3.30	1.41	133%
cold modify	12.09	4.56	165%
warm modify	15.64	3.83	308%
cold insert	46.67	40.79	14%
warm insert	50.50	40.41	25%

Table 3: Large database results

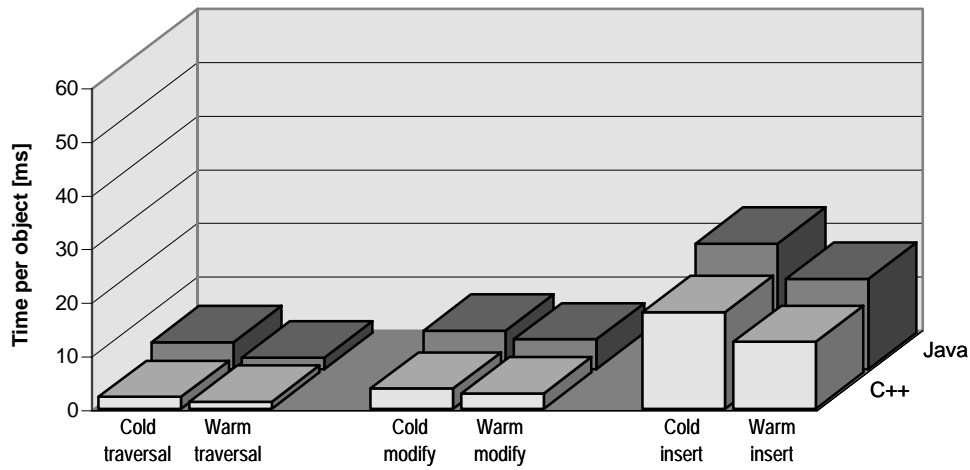


Figure 4: Small database results

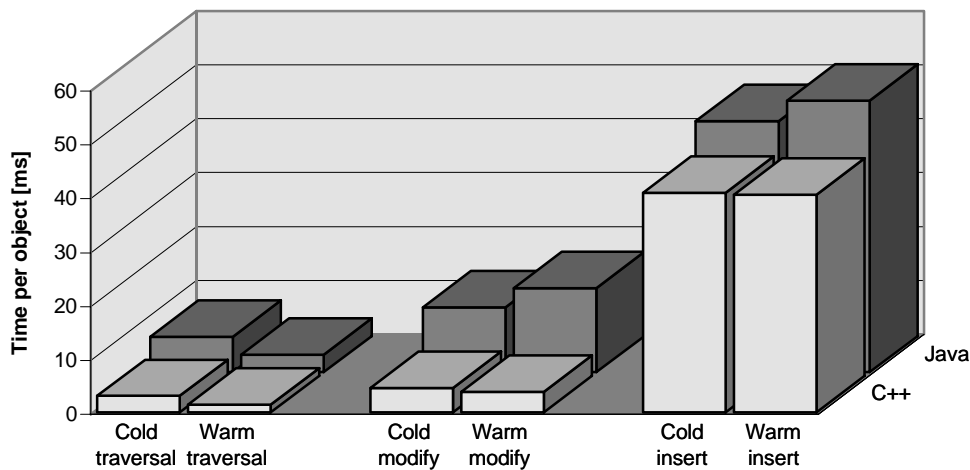


Figure 5: Large database results

The numbers presented in these tables are shown in a graphic form in figures 4 and 5, respectively for the small and for the large database.

The first thing to notice in these results is that Java is slower than C++. This is normal, given that we used an interpreting Java Virtual Machine. We expect that using a Just In Time Java environment will significantly reduce the differences.

An interpreted Java program is generally 5 to 10 times slower than the same program written in C++. Our measurements show execution times slower by 30% to 150% in most cases. This means that a substantial part of the time is spent in the database server and in the client libraries, and the work done in Java is relatively short.

The difference between cold and warm times is due to caching which improves the performances in the warm case. It is interesting to note the behavior of Java in the modify and insert experiments, in the large database case: the warm times are greater than the cold times. We do not have a valid interpretation for this case and we continue to investigate.

When comparing times of the small versus the large database, we notice a general increase of execution times. In the large database there is less locality than in the small database, which causes more pages to be read from disk. The modify time in Java has increased more than the modify time for C++ when going from the small database to the large database. This is a point which merits further investigation.

We are in the process of making performance measurements for comparing the JUB using O<sub>2</sub> versus JUB using a relational database.

## 7 Conclusion

We presented the main features of the Java Universal Binding, currently under development at O<sub>2</sub> Technology. We briefly discussed some implementation issues and presented performance results.

## References

- [1] M.P. Atkinson, L. Daynès, M.J. Jordan, T. Printezis, and S. Spence. An Orthogonally Persistent Java. *ACM Sigmod Record*, December 1996.
- [2] M.P. Atkinson, L. Daynès, M.J. Jordan, and S. Spence. Design Issues for Persistent Java: a type-safe, object-oriented, orthogonally persistent system. In *Proceedings of the Seventh Workshop on Persistent Object Systems*, May 1996.
- [3] F. Bancilhon, C. Delobel, and P. Kannelakis. *Building an Object-Oriented Database System - The Story of O<sub>2</sub>*. Morgan Kaufmann, 1992.
- [4] R. Cattell and J. Skeen. Object operations benchmark. *ACM Transactions on Database Systems*, 17(1), March 1992.
- [5] R.G.G. Cattell, editor. *Object Database Standard : ODMG - 93, Release 1.2*. Morgan Kaufmann, San Francisco, 1996.
- [6] A. Garthwaite and S. Nettles. Transactions for Java. *First International Workshop on Persistence and Java - PJI*, September 1996.
- [7] Graham Hamilton and Rick Cattell. *JDBC: A Java SQL API*, June 1996.
- [8] Marcin Skubiszewski and Patrick Valduriez. Concurrent garbage collection in O<sub>2</sub>. In *International Conference on Very Large Data Bases (to appear)*, 1997.
- [9] C. Souza dos Santos and E. Theroude. Persistent Java. *First International Workshop on Persistence and Java - PJI*, September 1996.