Accessing Warehoused Collections of Objects Through Java

S. Bailey, A. Goldstein, R. L. Grossman, and D. Hanley

Laboratory for Advanced Computing
University of Illinois at Chicago

Magnify, Inc.

September 6, 1996

Point of Contact: R. L. Grossman
grossman@uic.edu

Introduction

Persistence is important in a number of different domains including programming languages, internet-programming languages, object-oriented databases, and data warehouses. The Java language is rapidly growing in popularity and provides a number of attractive features, including relative simplicity, support for objects, cross platform portability, and security. In this position paper we discuss some of the issues arising when using Java to access warehoused collections of persistent objects.

Object-oriented databases are designed to support the storing and retrieval of objects. The underlying assumption is that objects will be frequently inserted and updated. Object warehouses are designed to support the analysis of large collections of objects. The underlying assumption is that the data is read often and occasionally appended or updated. In some broad sense, object-oriented databases are shaped by issues related to transactions and referential integrity while object warehouses are shaped by issues related to performance and indexing.

Recently, object warehouses have grown in importance with the rise of data mining. Data mining is the automatic discovery of patterns, associations and anomalies in data. Object warehouses provide a convenient foundation for data mining, especially the mining of complex data or data with complex relationships.

In this paper, we describe our initial design and implementation for a preprocessor to Java called JTool which extends the Java language to support the creation and access of object warehouses. This is based upon our previous work on a low overhead, high performance persistent object manager we developed called PTool which was primarily used to create object warehouses. JTool trades some of the simplicity and performance of PTool for the saftey, portability and security of Java.

We begin by examining how PTool currently creates object warehouses. We then explain our current approach to supporting persistent for Java.

PTool's current implementation

PTool is a light weight, high-performance persistent object manager that was developed by the

Laboratory for Advanced Computing at the University of Illinois at Chicago. PTool is optimized to provide low overhead access to data which is assumed to be read often, occassionally appended, and infrequently updated.

The C++ binding of PTool has a template object type known as a *Ref*. A *Ref* refers to an object in a 64-bit address space. By dereferencing a ref, PTool pulls this object into memory so that it can be accessed like an ordinary C++ object.

A ref object contains four numeric fields. These numeric fields refer to the Store, Folio, Segment, and offset of the object being referred to.

> A *Store* is a collection of folios, which are data files that contain persistent objects.
>
> A *Folio* is a data file of objects which is divided into smaller units called segments. Folios are the basic unit managed by the storage or file system.
>
> A *Segment* is a collection of contiguous objects. Segments are the basic units which are retrieved from disk and written into memory when a persistent object is dereferenced. Scanning warehousing collections of objects favors larger segments; accessing objects from a warehouse in random order favors smaller segments.
>
> The *Offset* describes the position of an object within a segment.

The following operations take place when a Ref is de-referenced:

1. the Store is opened, if necessary
2. the correct folio is opened
3. the segment is retrieved,
4. and the address of the object from inside if it is returned.

PTool employs a segment level cache in local memory so that de-referencing typically is only a fetch from a local memory structure. This has been optimized to an average case of only a handful of clock cycles.

Allocation in this scheme takes place only on the end of the heap. This means that allocation will always take place in a handful of cycles, and that complex object will retain good locality. The downside is that objects, once allocated, can never be freed. This means that objects created in a persistent space should either be objects that will need to persist as long as the data set is needed, or that the users should allocate temporary objects in a designated temporary store that will be periodically deleted. This preserves high allocation speed, heap compactness, and complex object locality.

De-refrencing a persistent object is all handled by normal C calls. Early versions of PTool exploited virtual memory to lower the overhead when accessing objects even more. To increase portability and reduce some bottlenecks this caused, PTool no longer uses this scheme.

To sum up, the merits of our scheme are speed, code simplicity, and portability. To gain this, we have sacrificed the ability to move our persistent objects accross platforms, the safety of transactions and concurrency support.

Obstacles to the interface

We cannot directly support the implementation describe above in Java for a variety of reasons:

- We do not have templates, or parameterized classes.
- We cannot overload operators, such as the -> operator, which is very important to the C++ binding of PTool.
- Java has far more stringent rules when casting pointers; we are not able to cast from a base binary type (such as characters) to a complex object type at all.

For these reasons, our current implementation of JTool requires extending the Java language by adding the keyword *persistent* and by allowing the overloading of some operators. We are writing a preprocessor which takes Java programs with these extensions and emits programs in standard Java. In contrast, PTool requires no extention to the C++ language and does not use a preprocessor; rather, it simply overloads existing operators.

Proposed Changes to the Java Language

In this section, we describe our proposed changes to the Java language.

Adding the keyword persistent *to the language.* When an object handle is being declared in persistent Java, by including the "persistent" keyword in the declaration, we will add a 64-bit reference (long int) into the class definition called "persistent_this" which will point to the objects location in the persistent address space. This must also be coupled with a persistent new to give us a handle to a persistent object. However, as we will see later, the compiler can fill in many of these attributes for us.

```
Example of "persistent" keyword use:

class Foo
{

   ...

}

public static void main( String args[] )
{
     persistent Foo   f;
     ....
}
```

Overloading the new *operator.* When allocating a persistent object, we require that the user specify the associated object warehouse by overloading the new operator. The compiler is allowed to provide this information when the allocation is to a child member of a persistent object; in this case, the child object will be allocated in the same store as its parent. This can be overridden by supplying the store information explicitly, to enhance performance.

```
Example of "new" operator use:

public static void main( Strings args[] )
{

    Store   s( "store1" );   // Open peristent addressing space
```

```
                                // see description of store class below

        persistent Foo f;       // declare peristent reference of type Foo

        f = new( s ) Foo;       // allocate new Foo within peristent addressing
                                //   space s.
        ...

    }
```

Overloading the assignment operator(=). We need to access the methods and attributes of persistent objects as if they were ordinary objects. To this end, we overload the assignment operator so that during a persistent reference assignment we can create a temporary transient copy of the peristent object on which all dot (.) operations will be valid. At persistent reference reassignment or program termination, we flush all attributes of the object to the peristent addressing space and allow the Java garbage collector to clean up the allocated transient space.

Overloadeding the dot operator(.). When the right-hand of the dot operator is a persistent reference as in the case of accessing persistent reference members of objects, the new dot operator will dereference the persistent object.

We also need to have a few classes present that will allow us to work with persistent objects. We need a `Store` class that will act as the handle into the persistent object space, and we need `Set` and `Iterator` classes that will allow us to extract objects from the root set of the persistent object space during the next program run. Finally a static instance of the Ptool class will contain all methods to fetch and flush primitive types from the persistent space.

There are various other sundry methods and objects, but the above comprises the visible syntax changes to the Java language. We feel it is important to make as few syntax changes as possible, both to ease the job of implementation, and to make sure new features of the Java language will not collide with our preprocessor. Other objects associated with the persistent object manager, are coded entirely in Java.

With this proposal, dereferencing a persistent reference actually occurs at assignment. During de-referencing, the Store, Folio, and Segment specified by the persistent address are fetched (as in PTool). However, unlike PTool, in the current implementation of JTool, a transient version of the object is allocated, and the transient copies attributes are updated to their persistent values. We are also experimenting with variants of this implementation.

JTool, especially in the proposed Java binding, attempts to provide persistency in as a transparent fashion as possible. It is difficult if not impossible to tell code manipulating persistent structures from code manipulating transient ones. This is important because we have programmer familiarity on our side, as well as the potential to utilize an already existent code base.

Implementation Proposal

Because of our emphasis on performance, it is unlikely that we could modify the Java Virtual Machine to suit our objectives. Because our implementation allows mixed persistent and transient objects, we must utilize symbols in the Java source; therefore processing the generated bytecodes is insufficent. Modifying the Java compiler ties us to one platform and makes the product less available to those on other platforms.

Our plan is to implement a preprocessor which will analyze Java source, extract out persistent information from our persistency keywords, and emit Java source which will then be fed to the Java compiler. The preprocesor adds `persistent_encode` and `persistent_decode` method to every class that is to be persistent. These methods will flush and fetch respectively the persistent attributes of the class.

Here is an example of PJava code before and after preprocessing:

```
PJava Code:

class Foo
{

   int Int1
   float Float1


}



class Populate
{

  public static void  main( String args[] )
  {

    Store s( "test1" );  // open a persistent space

    persistent Foo f;    //declare a persistent reference f


    f = new( s ) Foo;    //allocate a persistent instance of Foo
    f.Int1 = 10;          //Set attribute Int1
    f.Float1 = 20.5;        //Set attribute Float1

    s.insert_element( f );  //add object that f is pointing to to basic collecti
                            //of Store s

    f = new( s ) Foo;   //allocate another persistent instance of Foo
    f.Int1 = 11;         //Set attribute Int1
    f.Float1 = 21.3;     //Set attribute Float2

    s.insert_element( f );

  }
}


Java source target of above PJava source:

class Foo
{
   //PJAVA class addition:
   long persistent_this;      //holds persistent addresss of object

   int Int1
   float Float1

   //PJAVA class addition:
   //persistent_encode() flushes all attributes except "persistent_this" to the
```

```
   void persistent_encode()
   {
       Ptool.PutInt( persistent_this + 0, Int1 );
       Ptool.PutFloat( persistent_this + 4, Float1 );
    }

   //PJAVA class addition:
   //persistent_decode( long pptr ) fetches all attributes except "persistent_th
   //persistent space address of pptr and sets persistent_this to pptr.

    void peristent_decode( long pptr )
    {
       Int1 = Ptool.FetchInt( pptr );
       Float1 = Ptool.FetchFloat( pptr + 4 );
       persistent_this = pptr;
    }
}


class Populate
{

  public static void  main( String args[] )
  {

    Store s( "test1" );  // open a persistent space

    //PJAVA: persistent Foo f;    //declare a persistent reference f
    Foo f;  //declare a transient Foo with a "persistent_this" field



    //PJAVA: f = new( s ) Foo;    //allocate a persistent instance of Foo
    f = new Foo;  //make a new transient Foo
    f.persistent_this = s.malloc( 8 );   //allocate persistent space and set the
                                         //persistent_this pointer in f

    f.Int1 = 10;          //Set attribute Int1
    f.Float1 = 20.5;        //Set attribute Float1

    //PJAVA: s.insert_element( f );  //add object that f is pointing to to basic
                               //of Store s
    s.insert_element( f.persistent_this );  //pass the persistent_this pointer t
                                            //collection class



    //PJAVA: f = new( s ) Foo;   //allocate another persistent instance of Foo
    f.persistent_encode();   // at f reassignment flush contents of f to persist

    f = new Foo;   //make new transient Foo (old transient Foo will be handled b
                   //the Java garbage collector.

    f.persistent_this = s.malloc( 8 );  //allocate persistent space and set the
                                        //persistent_this pointer in f


    f.Int1 = 11;         //Set attribute Int1
    f.Float1 = 21.3;     //Set attribute Float2

    //PJAVA: s.insert_element( f );

    s.insert_element( f.persistent_this );  //pass the persistent_this pointer t
                                            //collection class
    f.persistent_encode();  //at program termination flush the contents of all p
```

```
                //to persistent space.

  }
}
```

Status

We have begun experimenting using this design by implementing the basic design "by hand" and evaluating the performance. We are currently working on a preprocessor to automate this process.

For more information

Please contact Robert Grossman, grossman@uic.edu.