JTool: Accessing Warehoused Collections of Objects with Java[1]

S. Bailey and R. L. Grossman[2]
Laboratory for Advanced Computing
University of Illinois at Chicago

July 25, 1997

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

Table of Contents

Abstract

The purpose of the work describe here is to gain experimental experience with data warehouses for large collections of Java objects. We report on the design, architecture, and early experimental work with a software tool called JTool for creating data warehouses of Java objects. Our primary interest is in building distributed data warehouses containing large collections of Java objects as a basis for the data mining of objects on the web. This work is broadly based upon our prior work with a software called PTool which we have used for the data mining of large collections of C++ objects in clustered computing environments [Grossman 1996 and 1997a].

---

[1] This research was supported by Grants from the National Science Foundation and the Department of Energy.

[2] Robert Grossman is a also a member of the technical staff at Magnify, Inc.

With Version 0.2 of JTool, we have built Gigabyte size data warehouses of Java objects and showed that JTool scales linearly with the size of the warehouse and the size and complexity of the underlying objects. Unfortunately, due to the overhead of the Java Virtual Machine and to our use of object serialization supported by JDK 1.1.1, querying a gigabyte warehouse of Java objects takes approximately 15 hours (vs minutes using PTool).

1 Introduction

Object warehouses are data management systems designed to support the analysis of collections of objects. Where as databases are designed for transactions environments with many writes, data warehouses are designed for analysis environment with many reads. Data warehouses speed up the analysis of data by precomputing and indexing as much derivative and summary data as feasible. At one end of a spectrum are transaction systems supporting updates of simple normalized relational data; at the other end are object warehouses supporting complex queries on complex objects which incorporate derivative and summary data.

Data mining is the discovery of patterns, associations, and anomalies in large data sets. It is convenient to divide data mining systems into three generations [Grossman 1997b]. Most systems today are first generation systems which support one or more data mining algorithms and which interface to file systems and databases. Second generation data mining systems are characterized in part by incorporating data management techniques in order to handle large data sets and to mine data out of memory. Third generation data mining systems support the mining of distributed data, including web-based data. Broadly speaking, there are two approaches to building third generation systems: one based upon agent-based computing and one based upon network-based computing. With Java emerging as one of the de facto standards for network-based computing, it is reasonable to explore the feasibility of designing third generation data mining systems using Java. This paper describes work in this direction.

The purpose of the work describe here is to gain experimental experience with large data warehouses for collections of Java objects. In this paper, we report on the design, architecture, and early experimental work with a software tool called JTool for creating data warehouses of Java objects. Our primary interest is in building data warehouses containing large collections of Java objects. This work is broadly based upon our prior work with a software called PTool which we have used for the data mining of terabyte size collections of C++ objects in clustered computing environments [Grossman 1996 and 1997a].

With this (second) version of JTool, we have built Gigabyte size data warehouses of Java objects and showed that JTool scales linearly with the size of the warehouse and the size and complexity of the underlying objects. Unfortunately, due to the overhead of the Java Virtual Machine and to our use of object serialization supported by JDK 1.1.1, querying a gigabyte warehouse of Java objects takes approximately 15 hours (vs minutes using PTool). Even so, because of the improvements expected in Java, we feel that the type of light weight data management employed by JTool for collections of Java objects may emerge as a viable foundation for third generation data mining systems, a topic we are currently exploring.

Our goals in the design of JTool are broadly similar to the design goals for PTool [Grossman 1995a and 1995b]:

1. We wanted a lightweight data management tool optimized for the types of queries common in data warehouses. Data warehouses are optimized for read-only queries and precompute and index as much data as possible in order to speed the performance of common queries.
2. We wanted the tool to scale to large collections of objects, to objects containing large numbers of attributes, and to queries which are numerically intensive.

3. We wanted the tool to support hierarchical storage systems incorporating memory, disk and tape and a variety of network protocols.

Broadly speaking, we designed a software tool which met these goals and tested it on a several data mining and data intensive applications with which we have worked in the past. We report on the preliminary results here. We emphasize that our intention was not to design a persistent version of Java - this has been done by others [Atkinson et al. 1996a, Atkinson et al. 1996b, Dearle et al. 1996, Garthwaite and Nettles 1996, and Jordan 1996].

It is well known that object serialization can be very efficient, especially for large collections of objects. A standard approach for dealing with this type of inefficiency is to partition hierarchically collections of objects into extents of increasing large size and manage each level in the hierarchy with a separate cache manager. This is the approach we take in this work. We present some evidence that this approach might prove useful in this context.

2 Related Work

Recently there has been a substantial work devoted to adding persistence to Java. Broadly speaking, there are three approaches. One approach to adding persistence is to provide direct support for persistence by changing the Java compiler or the Java virtual machine. Alternatively, a preprocessor for the Java source code could be used or a post-processor for the Java byte code. This has the advantage that it is the most powerful, but the disadvantage that it requires the most work. Another approach is to use existing data management systems to manage persistent data, either relational, object-oriented, or object-relational databases. This has the advantage that databases are widely available but the disadvantage that moving data between the two systems usually requires additional application code. Finally, the Java development environment itself supports a mechanism called serialization [Sun 1996] for taking a complex object and transforming it into a byte stream, which can then be made persistent. This has the advantage that it is well integrated into the Java environment, but the disadvantage that it can be very inefficient. We now discuss each of these approaches in greater detail.

2.1 Directly Supporting Orthogonal Persistence

In some sense the "right" way to provide persistence to Java is well known and articulated in [Atkinson et al. 83 and Atkinson and Morrison 95] through three principles. The first principle is that persistence should be orthogonal to type in the sense that all data whatever its type should have equal rights to persistence. The second principle is the principle of persistent independence which argues that all code should have the same form independently of how long the data upon which it acts persists. The third principle is the principle of transitive persistence which argues that whether data persists or not should be determined by whether the data is reachable from a persistent (root) object or not.

A difficulty with this approach is that providing persistence compliant with these three principles is not easy and requires changing the Java compiler, the Java virtual machine, preprocessing the Java source code, or post-processing the Java byte code. Moss and Hosking [Moss and Hosking 1996] classify some of the different approaches for providing orthogonal persistence to Java. Despite the effort required, there have been several implementations of persistent programming languages following these principles. A version of the Java language called PJava which supports persistence compatible with these three principles is described in [Atkinson et al. 1996 and Jordan 1996]. Another orthogonally persistent Java has been developed by Garthwaite and Nettles [Garthwaite and Nettles 1996]. Dearle et al. have developed an orthogonally persistent Java on top of their persistent operating system Grasshopper [Dearle et al. 1996]. Related material from this point of view is contained in [Morrison et al. 1996].

Malhotra argues in [Malhotra 1996] that scalability issues necessitate supplementing defining persistent objects through reachability with an alternate mechanism such as using explicit adds and deletes.

2.2 Interfaces to Databases and File Systems

There is embedded SQL interface between Java and relational databases called JDBC [Hamilton and Cattel 96]. This has the standard problems -- the impedance mismatch between the data models in the two different environments requires substantial additional programming -- and the standard advantages -- it is extremely useful due to the large amount of relational data in current systems. Related work is described in [Santos and Theroude 1996].

Most of the commercial object oriented database vendors have developed interfaces between their databases and Java. The ODMG is developing a standard for this interface [Cattel 1996]. Since the ODMG data model and the Java data model are similar, there is not the impedance mismatch problem that occurs with the JDBC binding. Most of these implementations appear to support orthogonal persistence.

An even greater amount of data is contained in legacy file systems than is contained in databases. Gruber [Gruber 1996] argues that this type of data is best accessed with external object faulting mechanisms.

2.3 Object Serialization

A serialization of a complex object is a byte stream representation of it. Alternate terms for this process are flattening and pickling. Once an object has been serialized, it can easily be made persistent, either using a file system or database. A common approach is to store serialized objects in a relational database as a BLOB or string. Another use for object serialization is that it provides an easy means to move complex objects between machines.

A closely related technique is to pack complex data into a string or blob. This is important for some high performance decision support and data mining applications. Accessing complex data with a conventional database might require several database accesses. By packing the data, this can be reduced to one operation. The trade-off is that updating the data is more difficult and expensive.

Perhaps the most important reason for employing object serialization to add persistence to Java is that it is easy: the Java Development Kit JDK 1.1.1 provides direct support for object serialization. The main disadvantage is that currently employing object serialization can be very inefficient. Employing object serialization also violates one of the three principles in [Atkinson et al. 83 and Atkinson and Morrison 95]. See [Atkinson et al. 1996b].

3 Using JTool

3.1 API Objects and Utilities

JTool Version 0.2 has three core objects for creating and manipulating large, persistent object stores: Ref, Store, and JTool. There is also a collection object for managing sets of persistent objects called: PSet. Finally, a utility called JTool_Register is required to use the JTool API with general objects.

3.1.1 Ref

A Ref object is a 64 bit reference to an object within a given name space. By name space we mean a collection of Stores. The Ref is broadly modeled after the ODMG specification of the same name; however, the lack of templates and operator overloading required us to change the syntax and exclude type information in Ref.

Ref Methods:

```
Ref();
```

The `Ref()` constructor available to the programmer takes no arguments and constructs a "NULL" Ref. By NULL, we mean that the Ref instance is not pointing to a legal position in the store.

```
Object Deref();
```

The `Deref()` method takes no arguments and returns the Object that is located at the persistent address in the persistent name space that the instance of Ref is pointing to. `Deref()` throws an exception if the instance is referencing an illegal address in the name space (e.g.. a Store that does not exist).

```
void Persist();
```

The `Persist()` method takes no arguments and has a void return. `Persist()` causes the persistent image of the object to become consistent with the current transient object to which the instance of Ref is pointing.

3.1.2 Store

A store object is a named persistent space into which persistent objects can be allocated and out of which persistent objects can be retrieved and modified. The store is also a collection class and supports API methods which allow functionality similar to that of the ODMG Bag class (i.e. allows insertion of duplicate objects). We have included this functionality in order to be compatible with the PTool-v2.3 API. However, future releases of JTool will most like only include a method to add on object to the root of a Store.

Store Methods:

```
Store( String );
```

The `Store( String )` constructor takes a String and opens the persistent space of that name. If the store has not be created before, the constructor initializes the store and adds the new store to the name space.

```
void insert_element( Ref );
```

The `insert_element( Ref )` method takes a Ref object as an argument and adds the reference to the Store's collection of objects. This method throws an exception if the Ref cannot be added to the collection.

```
boolean Next( Ref );
```

The `Next( Ref )` method takes a Ref object as an argument, attempts to set the Ref to point to the next object in the Store's collection and returns a boolean. This method returns true if

another object was available in the collection and false if the end of the collection has been reached and there are no more objects to return. By "next' object we mean in reference to previous calls to `Next()` starting with the first object being returned on the first call to `Next()`.

```
void Reset();
```

The `Reset()` method takes no arguments, returns void, and insures that the next call to the `Next()` method will attempt to return the first object in the Store's collection.

3.1.3 JTool

The JTool object handles persistence transactions on the stores in a given name space.

JTool Methods:

```
static Ref New( Store, Object );
```

The `New()` method is static, takes both a Store and an Object as arguments, attemps to allocate the object into the persistent Store and returns a Ref pointing to the persistent location of the object in the name space.

```
static void FinalizeJTool();
```

The `FinalizeJTool()` method must be invoked at the end of a JTool application to bring the object store into a consistent state.

3.1.4 PSet

The PSet is an untyped collection class and supports API methods which allow it functionality broadly similar to the ODMG Bag class (i.e. allows duplicates to be inserted). The name "PSet" is a carry over from the PTool-v2.3 API and will be mostly changed to PBag in future releases.

PSet Methods:

```
PSet( Store );
```

The PSet constructor available through the JTool API takes a Store as an argument and instanciates the PSet. The Store given in the constructor denotes the particular Store where new PSet "nodes" will be allocated. By "node" we mean an object which contains two Refs. One Ref points to an object in the collection and the other Ref points to the next "node" as in a linked list. It should be noted that the Store where "nodes" are allocated does not have any bearing the Store(s) where objects managed by the PSet are located.

```
void insert_element( Ref );
```

The `insert_element( Ref )` method takes a Ref object as an argument and adds the reference to the collection of objects. This method throws an exception if the Ref cannot be added to the collection.

```
boolean Next( Ref );
```

The `Next( Ref )` method takes a Ref object as an argument, attempts to set the Ref to point to the next object in the collection and returns a boolean. This method returns true if another object was available in the collection and false if the end of the collection has been reached and there are no more objects to return. By "next' object we mean in reference to previous calls to Next() starting with the first object being returned on the first call to `Next()`.

```
void Reset();
```

The `Reset()` method takes no arguments, returns void, and insures that the next call to the `Next()` method will attempt to return the first object in the collection.

3.1.5 JTool_Register (Java Application)

The JTool_Register application calculates and registers the "sizeof" objects which are to be made persistent in a given name space. Therefore, all objects which are going to be persistent must be registered with JTool_Register.

```
Usage:

java JTool_Register <object list>
```

3.2 An Example

The purpose of this section is to give a simple example of working with scientific data using JTool. The example in this section is adapted from [Grossman 1994], where further details can be found. There are three main steps to create and access a store of events: first, a schema for the store is designed, which defines the objects and their attributes; second, a store is created and populated with objects; and third, the store is queried for objects meeting specified criteria.

1. The first step is to define and register the schema for the store by defining the objects and their attributes. Simply defining the relevant Java classes, as in Event.java, Jet.java, and Lepton.java found in the appendix does this. It should be noted that most sub-objects (e.g. as Lepton is to Event) should be declared with a Ref in the main object. All objects that are to be persistent must implement Serializable as per Object Serialization in JDK 1.1.1 which simply means adding the words "`implements Serializable`" to the class definitions. This is necessary since JTool currently utilizes Object Serialization found in JDK 1.1.1 to cast byte streams to objects as discussed in section 4.2.

To register one would execute the following command:

```
> java JTool_Register Event Jet Lepton
```

2.The second step is to populate the store. The statement

```
Store store = new Store("PsiEvents");
```

creates a store with the internal handle a, and the external name PsiEvents, or opens the store for appending if it already exists.

When a persistent object is created in a Store, it must also be part of some kind of data structure that allows access to it in the future. JTool comes with a built-in sequential data structure as part of the Store and an external version call PSet.

To make an object persistent, the standard Java statements

```
Event event;
event = new Event();
```

are replaced by

```
Ref  eventRef;
eventRef = JTool.New( store, new Event() );
```

In order to access the object later, it must be added to the base collection in the store with the statement:

```
store.insert_element(eventRef);
```

For a working example, see the source code Pop.java in the appendix.

Attributes for persistent objects can be accessed in the follwing ways:

```
((Event)eventRef.Deref()).vertex = 10.4;
```

or

```
Event event = (Event)eventRef.Deref();
event.vertex = 10.4
```

assigns 10.4 to the vertex attribute of the Event object.

3.The third step is to query the persistent store of objects. Any other process can open the store with the external name PsiEvents and access its elements using the internal handle b with the statement

```
Store store = new Store("PsiEvents");
```

To loop through all objects in a store:

```
Ref eventRef = new Ref();
while( store.Next( e ) )
{ . . . }
```

For a working example, see the source code Pop.java and Acc.java.

4 Design and Implementation

4.1 Physical Storage Management

Both JTool and PTool achieve scalability by hierarchically grouping objects into extents of increasing size: objects are gathered into segments, segments into folios, and folios into stores. A name space consists of a number of stores. The different size extents are then managed by a multi-level caching and migration system. For more details, see [Grossman 1995a].

Segment. A segment is a physical collection of objects. The size of the segment is currently set at 65K. Segments are the basic unit for transferring objects between disks and memory.

Folio. A folio is a collection of segments and the basic unit of managing segments on secondary and tertiary storage systems, such as disks and tapes. Folios are currently implemented as files. After a

network Waddle has been implemented, we will be able to distribute the folios of a Store across an arbitrary number of nodes in a network and create stores which are larger than the maximum file size allowed by a single file system.

Store. A store is a collection of folios.

Name Space. A name space is a collection of stores for which there is a JTool Registry and a JTool Database Map. Complex persistent objects can contain sub-objects, which may reside in other stores as long as all the stores in question are in the same name space.

In summary, we designed JTool to use four storage levels for performing physical data management. With this design we can theoretically create and access very large object stores, and yet manage the store efficiently. This is analogous to multi-level caching schemes, which are common in distributed file systems.

4.2 The Use of Object Serialization

The current release of JTool utilizes the Object Serialization support in JDK 1.1.1 to facilitate the casting of bytes held in the JTool Cache to objects and vice-versa. JTool itself, however, provides a much greater functionality than simple Objects Serialization. A JTool name space functions as a randomly accessible, 64bit addressable, persistent heap as opposed to a flat file. Objects can be much larger than the single file size limit and objects can span multiple Stores.

In order to retrieve or store objects, JTool fetches the appropriate segment into the cache as discussed below, creates a temporary ObjectInputstream or ObjectOutputstream at the appropriate offset in the segment were the objects resides or is to reside, and finally invokes `objectWrite( Object )` or `objectRead()` as necessary.

4.3 Architecture

In this section we will examine the internal architecture of JTool. Most notably we will better describe the Ref and outline some important modules including: The Database Map, Object Registry, Waddle (an unfortunate name retained from the PTool system), and Cache. Finally we will step through an example of allocating a new persistent object using JTool.

4.3.1 The Ref

The Ref, points to objects in the persistent space. Once the `Deref()` method is called or a `JTool.New()` is invoked, the Ref also points to a transient image of the persistent object. The attributes of ref are shown below:

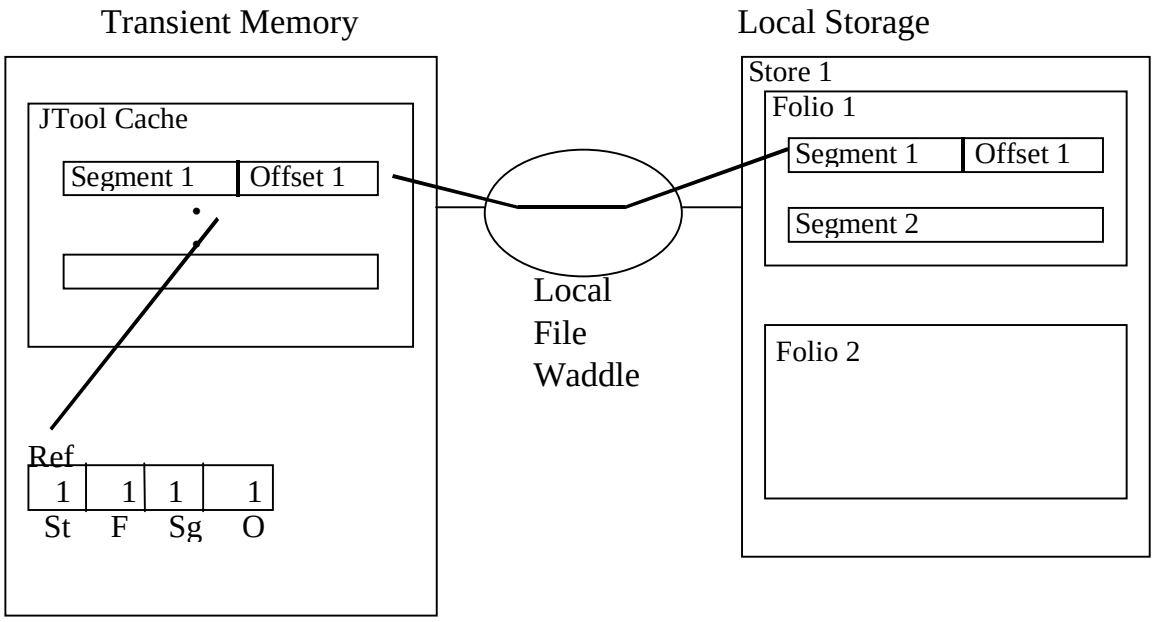
```
class Ref implements Serializable
{
      transient Object object;
      long ppointer;
}
```

The “object” attribute of the Ref is marked as transient for the case when a persistent object A has a Ref to a persisent object B as an attribute, as in:

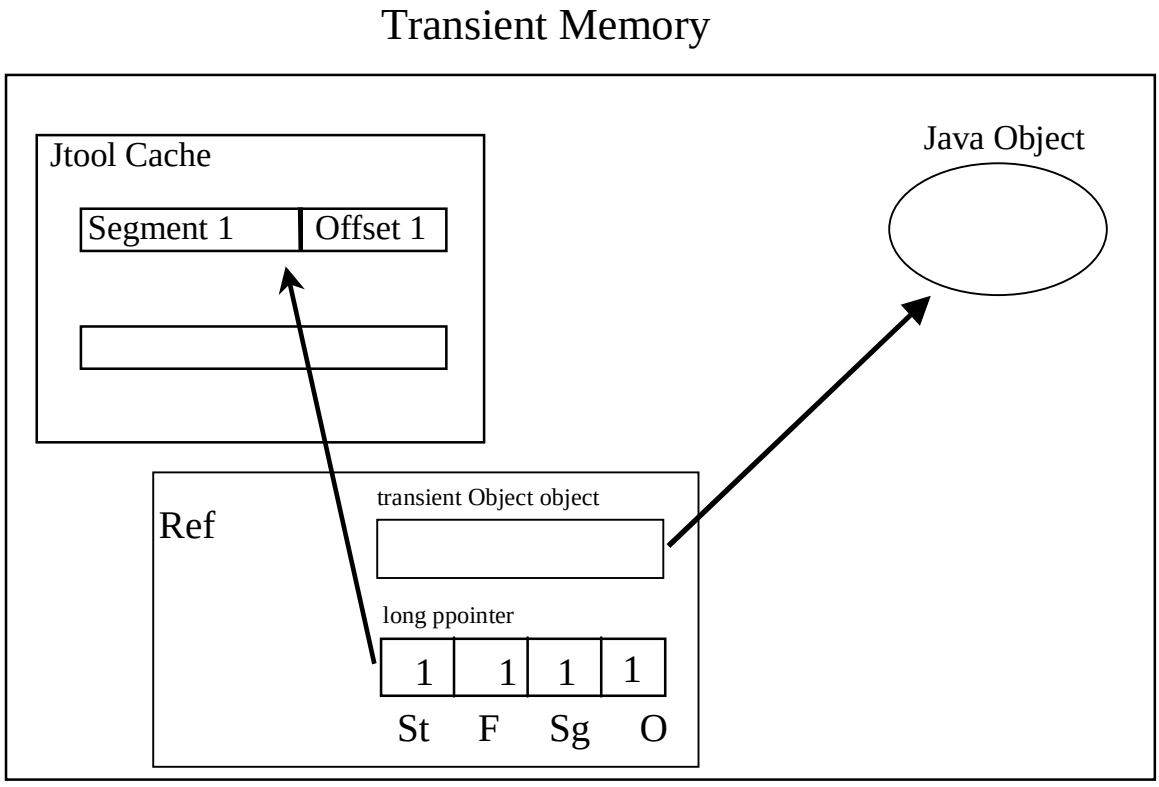
```
class A
{
      Ref b;
}
```

In this circumstance, only the persistent address (i.e. “ppointer” attribute of the Ref for object B) is stored with the object A. The Persist() method of the Ref for object B must be called to make object B persistent. This functionality is desired so that an object can be much larger than the memory limit provided by the transient 32bit addressing space of Java. For example, consider a very large collection of objects as an object itself, if the collection contained 100 Gigabytes of objects it would be required to use Refs since the whole collection cannot fit into transient memory a single time.

The ppointer attribute of a Ref is a long. These 64bits are further subdivided by JTool into four fields: Store, Folio, Segment, Offset. Currently, each field is 16bits long. These four fields are recovered from the long by bit shifting. The figure below shows how these are used.



After the Deref() method has been called on a Ref object and the appropriate segment and offset has been located in the cache, the byte array is “deserialized” into a Java object and “object” attribute of the Ref is updated to point to the new transient image of the persistent object.



4.3.2 Database Map (JTool.DbMap)

The Database Map maps the global identifiers of Stores to ID numbers used in Refs as the most significant 16bits of the Ref's ppointer attribute for objects in the a particular Store. Any time a new Store is created it is assigned a Store ID. The Store ID and the Store name (i.e. global identifier) mapping is added to the Database Map. The Stores contained in a particular Database Map define a name space. Any object in a particular Store can contain Refs pointing to objects in other Stores as long as the Stores in question are in the same name space (i.e. all the Stores are in the Database Map).

4.3.3 Object Registry(JTool.Registry)

The Object Registry maps class types which can be made persistent to a corresponding image size in the persistent Store. The Object Registry is initialized using the JTool_Register application. This effectively gives JTool a "sizeof" method which it can use to allocate appropriate space in the persistent store. The Object Registry is currently implemented as a Serialized Hashtable.

4.3.4 Waddle

The Waddle module of JTool (Waddle is a historical name from PTool) is responsible for the actual fetching of segments from secondary storage into the JTool cache. Currently, only a local file Waddle is provided with JTool. However, like PTool, we expect to have network, compressed file, and tape Waddles available for future implementations of JTool.

There is one instance of a Waddle for every open Store. The Waddle instance fetches segments from secondary memory by opening the appropriate folio (file), seeking to the appropriate offset in the folio (i.e. where the wanted segment starts), and reading or write the segment as requested.

4.3.5 Run-time Store and Waddle Tables

These are look up tables containing pointers to instances of Store and Waddle based on the Store ID.

4.3.6 Segment Cache

The Segment Cache holds a variable number of Segments (decided at compile time) in main memory during the lifetime of a JTool application. When there is a Cache "miss" (i.e. a particular segment is needed but not in the Cache), the Cache makes a call to the appropriate Waddle based on Store ID and requests the particular segment. The current cache eviction policy dumps the oldest segment (i.e. been in the cache the longest). Furthermore, every segment is flushed back to the persistent folios when evicted. We plan to add a differentiation of "dirty" and "clean" segments in future releases. We recognize the cache implementation as naïve as well as a potentially dominant performance factor.

4.3.7 An Example of Object Allocation

Below is a diagram of main memory after a new object Foo is allocated into Store "test" with the following code fragment. This also assumes Foo has been registered with JTool_Register application.

```
.
.
.
Store s  = new Store( "test" );
Ref pf ;
pf = JTool.New( s, new Foo() );
.
.
.
```

Here are the steps invovled:

Code: `Store s = new Store("test");`

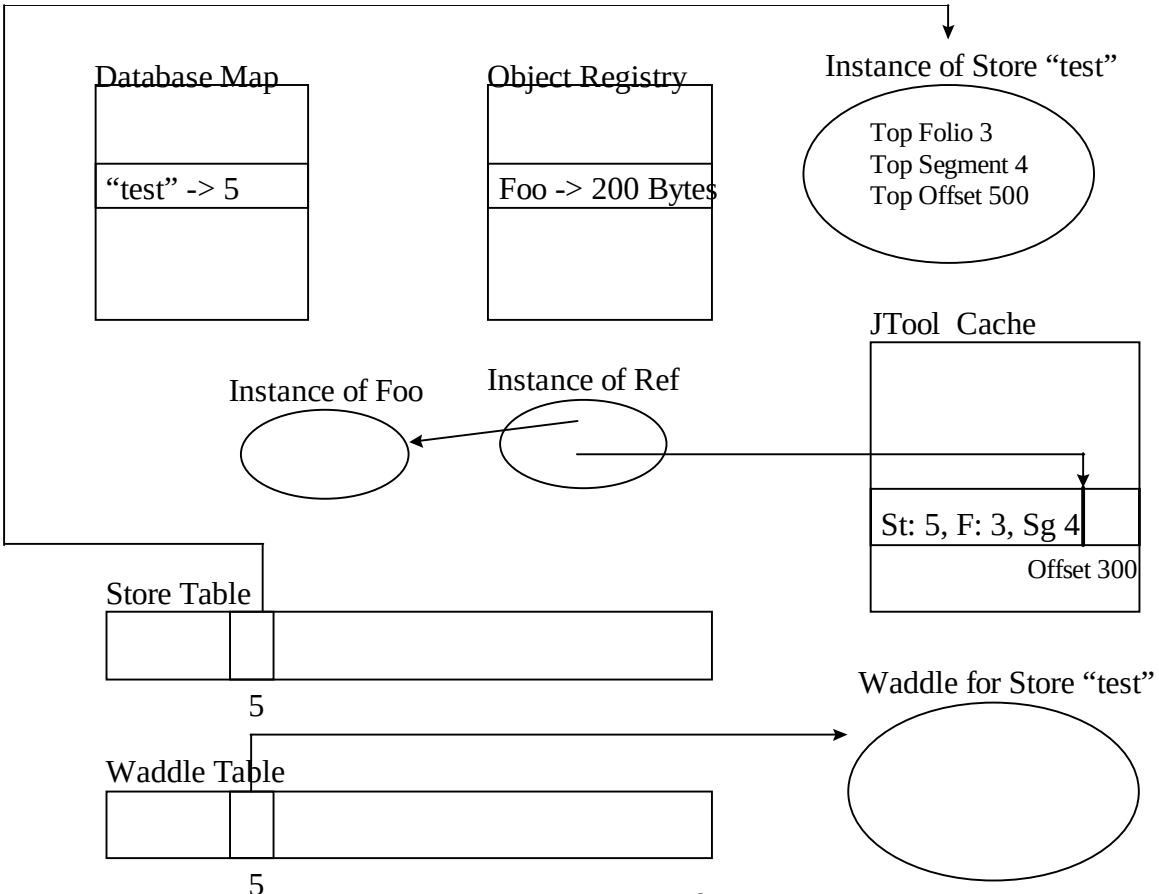
Step 1. Lookup Store ID in Database Map for "test"
Step 2. Initialize Store and Waddle instances and set Store and Waddle Lookup Tables to point to them. In this example, the Store "test" has a Top Folio of 3, Top Segment of 4, Top Offset of 300. These three numbers denotes the current end of the Store; new objects will be allocated after that.

Code: `Ref pf;`

Step 3. Make a Ref variable. Currently it is null.

Code: `pf = JTool.New( s, new Foo() );`

Step 4. Make a new instance of Foo
Step 5. Make a new instance of Ref and set the "object" attribute to point to the new instance of Foo.
Step 6. Look of "sizeof" Foo (i.e. 200 Bytes) in the Object Registry.
Step 7. Set Top Offset of Store "test" to 500 (i.e. allocate 200 Bytes in the Store).
Step 8. Set the "ppointer" attribute of the Ref to [5, 3, 4, 300] (i.e. Store 5, Folio 3, Segment 4, Offset 300).
Step 9. Request the following Segment: Store 5 ("test"), Folio 3, Segment 4.
Step 10. If not present in the Cache, the Waddle for "test" fetches the segment into the Cache.
Step 11. Create a temporary ObjectOutputstream 300 Bytes into the Segment.
Step 12. Call writeObject(Object) on the ObjectOutputstream passing it the new Foo object.
Step 13. Return the new Ref and set pf equal to it.



Run-time JTool Memory Image After Object Allocation

4.4 Status

JTool Version 0.2 was used for the experimental studies reported here. JTool is based upon Version 2.3 of PTool.

5 Experimental Results

5.1 Experimental Facility

Experiments were conducted on a Sun Sparc 20 with 32 Mbytes of RAM running Solaris 2.5.1 and Jdk 1.1.1. The disk on which we populated stores was a 9 Gig Seagate NFS mounted to the SparcStation over standard ethernet. The version of JTool used in the tests was JTool Version 0.2.

5.2 Data Sets

For ease of comparison with our past work mining scientific data, we created a data set containing synthetic data called Events, broadly similar to data arising in high energy physics [Grossman 1995a and 1996]. More specifically, we populated a series of stores using JTool containing Events defined by the class listed in the appendix. The Stores ranged in size from 2 to 1000 Megabytes. The attributes of the events were assigned random values. Each event had 2 Lepton and 1 - 3 Jets as attributes.

We also populated a series of store keeping the number of Events constant but varying the number attributes (Jets) between 5 and 1000.

5.3 Tests

1. First, we measured the access time for examining all entire Event objects (i.e. including its Leptons and Jets) in a store as we varied the Store size.

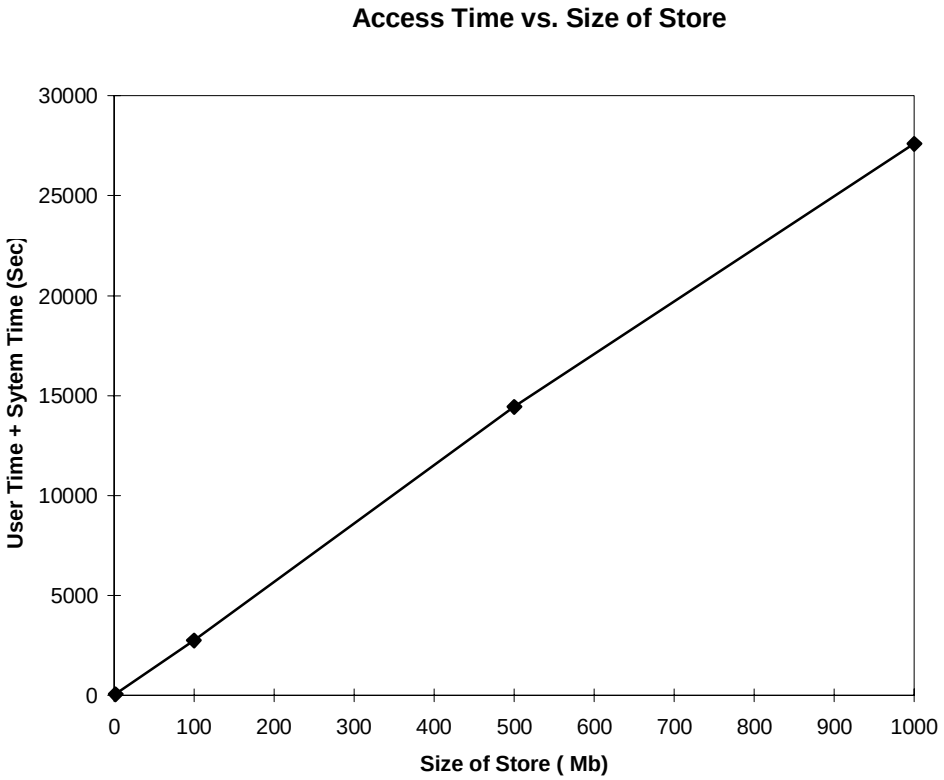
2. Second, we measured the access time for examining all entire Event objects in a store while varying the number of attributes per Event.

The data and tests are designed so that the best performance possible is a linear scale up as the amount of data and number of attributes increases. With inappropriate designs, linear scale up fails after a certain point.

5.4 Summary of Results

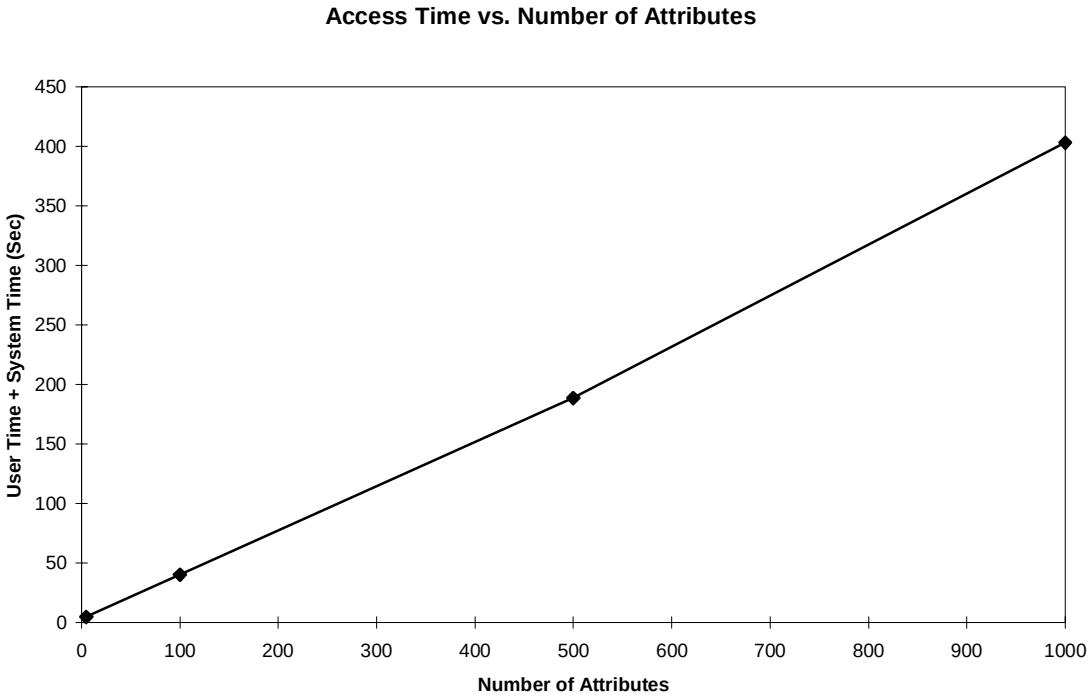
In our first test we found that access times did indeed scale linearly as a function of store size:

	ACCESS TIME (sec)		
STORE SIZE (Mb)	Real Time	User Time	System Time
2	176.13	57.12	1.2
100	5469.1	2726.51	22.12
500	31051.15	14321.51	116.33
1000	51671.77	27378.4	217.11



In our second series of tests we found that access times did indeed scale linearly as a function of the number of attributes:

	ACCESS TIME (Sec)		
Number of Attributes	Real Time	User Time	System Time
5	4.93	4.05	0.7
100	40.56	39.53	0.81
500	191.18	186.95	1.66
1000	534.36	400.37	2.69



6 Summary and Conclusions

In this paper, we have described the design and implementation of a software tool for creating data warehouses of Java objects called JTool and reported on experimental studies showing that JTool scales as designed for data warehouses containing up to one Gigabyte of data and for objects containing up to one thousand attributes.

The current release of JTool (Version 0.2) is our first pass with this design (the prior version had a very different design) and has room for considerable improvement. We are currently experimenting with JTool's caching and migration methods, the developments of alternatives to object serialization, and running additional applications and benchmarks.

Our main interest in JTool is as a data management infrastructure for data mining systems designed to mine collections of complex data distributed over local and wide area networks. With the rapid growth of the net, networked information is growing at a far faster pace than our ability to effectively use it. Much of this information consists of collections of Java objects. Data mining provides one means of automatically discovering patterns and associations in large data sets; without the development of software tools such as JTool, mining collections of Java objects will be more difficult.

7 References

[Atkinson et al. 83] M. P. Atkinson, K. J. Chisholm, W. P. Cockshott, and R. M. Marshall, Algorithms for a Persistent Heap, IEEE Software, Practice and Engineering, Volume 13, pages 259-272, 1983.

[Atkinson and Morrison 95] M. P. Atkinson and R. Morrison, Orthogonal Persistent Object Systems, VLDB Journal, Volume 4, pages 319-401, 1995.

[Atkinson et al. 1996a] M. P. Atkinson, M. J. Jordan, L. Daynes, and S. Spence, Design Issues for Persistent Java: a type safe, object oriented, orthogonally persistent system, Proceedings of The First International Workshop on Persistence and Java, Sunlabs Technical Report, http://www.dcs.gla.ac.uik/rapids/events/pj1.

[Atkinson et al. 1996b] M.P. Atkinson, L. Daynes, M. J. Jordan, T. Printezis, and S. Spence, An Orthgonally Persistent Java, SIGMOD 96.

[Bailey 1997] S. Bailey, A. Goldstein, R. L. Grossman, and D. Hanley, Accessing Warehoused Collections of Objects Through Java, Proceedings of The First International Workshop on Persistence and Java, Sunlabs Technical Report, http://www.dcs.gla.ac.uik/rapids/events/pj1.

[Cattel 1996] R. G. G. Cattel, Object Database Standard: ODMG-96, Release 1.2, Morgan Kaufmann, San Francisco, 1996.

[Dearle et al. 1996] A. Dearle, D. Hulse, and A. Farkas, Operating System Support for Java, Proceedings of The First International Workshop on Persistence and Java, Sunlabs Technical Report, http://www.dcs.gla.ac.uik/rapids/events/pj1.

[Garthwaite and Nettles 1996] A. Garthwaite and S. Nettles, Transactions for Java, Proceedings of The First International Workshop on Persistence and Java, Sunlabs Technical Report, http://www.dcs.gla.ac.uik/rapids/events/pj1.

[Grossman 1994] R. L. Grossman, ``Working With Object Stores of Events Using PTool," 1993 Cern Summer School in Computing, C.E. Vandoni and C. Verkerk, editors, CERN-Service d'Information Scientifique 94-06, pp. 66--97, 1994.

[Grossman 1995a] R. L. Grossman, N. Araujo, X. Qin, and W. Xu, ``Managing physical folios of objects between nodes," Persistent Object Systems (Proceedings of the Sixth International Workshop on Persistent Object Systems), M. P. Atkinson, V. Benzaken and D. Maier, editors, Springer-Verlag and British Computer Society, 1995, pages 217--231.

[Grossman 1995b] R. L. Grossman, D. Hanley, and X. Qin, ``PTool: A Light Weight Persistent Object Manager," Proceedings of SIGMOD 95, ACM, 1995, p. 488.

[Grossman 1996] R. L. Grossman, The Terabyte Challenge: An Open, Distributed Testbed for Managing and Mining Massive Data Sets, Proceedings of the 1996 Conference on Supercomputing, IEEE, 1996.

[Grossman 1997a] R. L. Grossman, S. Bailey and D. Hanley, Data Mining Using Light Weight Object Management in Clustered Computing Environments, Proceedings of the Seventh International Workshop on Persistent Object Stores, Morgan-Kauffmann, San Mateo, 1997.

[Grossman 1997b] R. L. Grossman, Data Mining: Challenges and Opportunities During the Next Decade, submitted for publication.

[Gruber 1996] O. Guber, Transparent Access to Legacy Data in Java, Proceedings of The First International Workshop on Persistence and Java, Sunlabs Technical Report, http://www.dcs.gla.ac.uik/rapids/events/pj1.

[Hamilton and Cattel 96] G. Hamilton and R. Cattel, JDBC: A Java SQL API, http://splash.javasoft.com/jdbc, June, 1996.

[Jordan 1996] M. Jordan, Early Experiences with Persistent Java, Proceedings of The First International Workshop on Persistence and Java, Sunlabs Technical Report, http://www.dcs.gla.ac.uik/rapids/events/pj1.

[Malhotra 1996] A. Malhotra, Persistent Java Objects: A Proposal, Proceedings of The First International Workshop on Persistence and Java, Sunlabs Technical Report, http://www.dcs.gla.ac.uik/rapids/events/pj1.

[Morrison et al. 1996] R. Morrison, R. Connor, G. Kirby and D. Munro, Can Java Persist?, Proceedings of The First International Workshop on Persistence and Java, Sunlabs Technical Report, http://www.dcs.gla.ac.uik/rapids/events/pj1.

[Moss and Hosking 1996] J. E. B. Moss and A. L. Hosking, Approaches to Adding Persistence to Java, Proceedings of The First International Workshop on Persistence and Java, Sunlabs Technical Report, http://www.dcs.gla.ac.uik/rapids/events/pj1.

[Santos and Theoude 1996] C. S. dos Santos and E. Theroude, Persistent Java, Proceedings of The First International Workshop on Persistence and Java, Sunlabs Technical Report, http://www.dcs.gla.ac.uik/rapids/events/pj1

[Sun 1996] Sun Microsystems, Inc., Java Object Serialization Specification, Draft Revision, 0.9, http://chatsubo.javasoft.com/current/doc/rmi-spec/rmiTOC.doc.html, 1996.

Appendix A. Event.java

```
import java.io.*;

class Event implements Serializable
{
    int runNumber = 0;
    int eventNumber = 0;

    double vertex = 0.0;

    Ref lepton1 = new Ref();
    Ref lepton2 = new Ref();

    Ref JetSet = new Ref();

    Event()
    {
    }
    Event( Store s )
    {
        runNumber = (int) ( Math.random() * 10000 );
        eventNumber = (int) ( Math.random() * 10000 );

        vertex = (double) Math.random();


        lepton1 = JTool.New( s, new Lepton() );

        lepton2 = JTool.New( s, new Lepton() );
```

```
        JetSet = JTool.New( s, new PSet( s ) );


        int numberofJets = (int)( ( Math.random() * 3 ) + 1 );
        for( int x = 0; x < numberofJets; x++ )
        {
            Ref j = JTool.New( s, new Jet() );
            ((PSet)JetSet.Deref()).insert_element( j );
        }

        JetSet.Persist();
    }

    public String toString()
    {
        String JetString = new String();
        Ref j = new Ref();
        PSet ps = (PSet)JetSet.Deref();
        while( ps.Next( j ) )
        {
            JetString = JetString + j.Deref();
        }
        return new String( "Run Number: " + runNumber +
                           "\nEvent Number: " + eventNumber +
                           "\nLepton 1: " + lepton1.Deref() +
                           "\nLepton 2: " + lepton2.Deref() +
                           "\nJet Set: " + JetString +

"\n=======================================\n");
    }
}
```

Appendix B. Lepton.java

```
import java.io.*;

class Lepton implements Serializable
{
    double p[] = null;
    double charge = 0.0;

    Lepton()
    {
        charge = (double) Math.random();
        p = new double[4];
        for( int x = 0; x < 4; x++ )
        {
            p[x] = (double) Math.random();
        }
    }

    public String toString()
    {
        String pstring = new String();
        for( int x = 0; x < 4; x++ )
        {
            pstring = pstring + new String( "\n    p[" + (x+1) + "]: " +
p[x] );
        }
        return new String( "\n  Lepton: " +
                           "\n    charge: " + charge +
                           pstring + "\n" );
    }
}
```

Appendix C. Jet.java

```
import java.io.*;

class Jet implements Serializable
{
    int jet_num = 0, ntrk = 0;
    double q_frac = 0.0, phi_jet = 0.0, eta_jet = 0.0, mass_jet = 0.0,
pt = 0.0;


    Jet()
    {
        jet_num = (int) (Math.random() * 10);
        ntrk = (int) ( Math.random() * 100 );

        q_frac = (double) Math.random();
        phi_jet = (double) Math.random();
        eta_jet = (double) Math.random();
        mass_jet = (double) Math.random();
        pt = (double) Math.random();
    }

    public String toString()
    {
        return new String( "\n Jet: " +
                           "\n    jet_num: " + jet_num +
                           "\n    ntk: " + ntrk +
                           "\n    q_frac: " + q_frac +
                           "\n    phi_jet: " + phi_jet +
                           "\n    eta_jet: " + eta_jet +
                           "\n    mass_jet: " + mass_jet +
                           "\n    pt: " + pt + "\n" );
    }
}
```

Appendix D. Pop.java

```
import java.io.*;


class Pop extends JTool
{
      public static void main( String args[] )
      {

        Store store = new Store( "PsiEvents" );

        Ref eventRef;

        try{
        for(  int a = 0 ; a < 1000 ; a++ )
            {
                eventRef = JTool.New(  s, new Event( s ) );
                store.insert_element( eventRef );
          }
            FinalizeJTool();
        }catch(Exception e ){ System.out.println( "Exception: " + e); }
      }
}
```

Appendix E. Acc.java

```
import java.io.*;

class Acc extends JTool
{
      public static void main( String args[] )
      {
        Store store = new Store( "test" );
        Ref eventRef = new Ref();
      try{
        while(  s.Next( eventRef ) )
        {
            System.out.println( eventRef.Deref() );
        }
        }catch(Exception e){};
        FinalizeJTool();
      }
}
```