Java™ Object Mapping and Persistence with Demand Paging

Master’s Thesis, 20 credits

Tutor: Gunnar Gunnarsson

Markus Mårtensson

13th February 2001
Abstract

This report makes a survey of database and mapping paradigms available for the Java™ platform, as well as available open source implementations thereof.

A new persistence scheme is introduced, in which virtual memory techniques are adopted for use from within the Java™ virtual machine. The garbage collector is used to automatically page objects to secondary storage, an approach which relieves the developer of having to deal with explicit database access.

The scheme allows objects to be temporarily removed from memory, thereby reducing overall memory consumption, without breaking the object reference graph. It also allows the state of a program to be made persistent between sessions.

An implementation is provided, proving that the scheme is feasible not only in theory, but also in practice. It is however concluded that the usability of the scheme is limited by its meager support for concurrent access and transactions.
# Contents

1 Introduction ........................................... 1

2 Goal and Outline of the Thesis ....................... 3

3 Databases .................................................. 5
   3.1 Categories .............................................. 5
      3.1.1 Hash files ......................................... 5
      3.1.2 Relational Databases ............................... 6
      3.1.3 Object Databases .................................. 6
      3.1.4 Object-Relational Databases ....................... 7
   3.2 Tools .................................................... 8
      3.2.1 PPHash ............................................. 8
      3.2.2 XL2 ................................................ 8
      3.2.3 MySQL ............................................ 9
      3.2.4 ozone ............................................ 10
      3.2.5 PostgreSQL ....................................... 11

4 Mapping .................................................... 13
   4.1 Categories .............................................. 13
      4.1.1 Serializable ....................................... 13
      4.1.2 Manual Mapping ................................... 14
      4.1.3 Introspection ..................................... 14
   4.2 Tools .................................................... 15
      4.2.1 Castor ............................................. 15
      4.2.2 JDO ................................................ 16
      4.2.3 jRelationalFramework ............................. 16

5 Garbage Collection ........................................ 17

6 Paging Theory ............................................. 21
   6.1 Demand Paging and Page Replacement ................. 21
   6.2 Object Paging ........................................... 21

7 Method ..................................................... 23

8 Results ..................................................... 25
   8.1 System Overview ....................................... 25
   8.2 User Guide ............................................. 26
   8.3 Pitfalls ................................................ 28
   8.4 Limitations ............................................ 29
   8.5 Problems ............................................... 29
## 9 Discussion

- 9.1 Implementation .......................................................... 31
- 9.2 Persistence Solutions .................................................. 31
- 9.3 Object Paging vs Demand Paging ................................. 31
- 9.4 Future ................................................................. 32

## 10 Acknowledgments

## Appendices

### A Source code

- A.1 Persistable.java ......................................................... 37
- A.2 World.java ........................................................... 42
- A.3 Node.java ............................................................ 44
List of Examples

1. Unstructured data. ................................................. 5
2. Structured data. .................................................... 5
3. Relational database. .............................................. 6
4. Using JDBC™ ....................................................... 7
5. Using PPHash ....................................................... 8
6. XL2 transactions. .................................................. 9
7. XL2 references. ................................................... 9
8. Making a MySQL connection using JDBC™ ................ 10
9. Using the ODMG interface of ozone. ......................... 10
10. Making a PostgreSQL connection using JDBC™ .......... 11
11. Serializable. .................................................... 13
12. Manual mapping. ................................................ 14
13. Accessors. ..................................................... 15
14. Reference types. ................................................. 17
15. Resetting references. ......................................... 18
16. Using a finalizer. ................................................ 19
17. Starting the sample program. ................................. 27
18. Walking the world tree. ...................................... 27
19. Adding to a node’s description. ............................... 27
20. State persistence between sessions. ........................ 28
21. The dangers of non-static inner classes. .................. 28
22. Using static inner classes. .................................... 29

List of Figures

1. Object reachability. .............................................. 17
2. Weak reference active. ......................................... 18
3. Weak reference cleared. ..................................... 18
4. UML diagram. .................................................. 25
5. World tree. ..................................................... 26
1 Introduction

People, when asked what they dislike about Java™, tend to state lack of speed and excessive memory consumption as the main issues. The former becomes less true with each passing day as JVM engineers work their magic; memory usage is however not likely to decrease at a comparable rate.

My personal experience with large Java™ systems is that the majority of objects are in a dormant state. For long periods of time, these objects do nothing but consume memory.

A virtual machine that hogs too much memory resources will inevitably be paged out to disk by the underlying operating system, which has little or no understanding of the inner workings of the JVM. This situation may lead to thrashing, i.e a state where more time is spent paging than executing.

Explicit database coupling is often used to reduce the number of objects in memory, a technique which breaks the natural object graph view of the data and thereby makes reference traversal impossible. An alternative solution is called for.
2 Goal and Outline of the Thesis

I propose a paging system to be used from within the JVM, implemented using a persistence layer with garbage collection hooks. Memory will seem infinite and object graphs will be kept intact, but the bulk of the objects will actually be kept on disk.

In order to put the proposal in proper context, this thesis will also make a survey of available database paradigms and implementations for the Java™ platform, located in section 3. In section 4, methods for mapping between objects and databases are discussed. A brief explanation of the garbage collector is available in section 5, followed by a few notes on paging in section 6. In sections 7 and 8, I will present my method and results respectively. The discussion of section 9 concludes the report.
3 Databases

A database is usually defined as a structured collection of related data. Data is of course synonymous to facts or pieces of information. Just about any document, take for example an owners manual for a television set, is filled with related facts. Few would however label the manual as a database. Consider the following related facts written in prose:

John Smith, the mechanic, has just turned fifty. His daughter Jane, which was born shortly after his 23rd birthday, is currently studying at Harvard.

Example 1: Unstructured data.

This information is not sufficiently structured for a today's computer technology to allow searches and present distinct, correct, and complete facts as result. Example 2 shows a commonly used structured approach.

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Occupation</th>
<th>Father</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Smith</td>
<td>50</td>
<td>Mechanic</td>
<td>John Smith</td>
</tr>
<tr>
<td>Jane Smith</td>
<td>27</td>
<td>Student</td>
<td></td>
</tr>
</tbody>
</table>

Example 2: Structured data.

A database management system is a set of programs which provides means of creating and managing databases. There are many DBMS paradigms and implementations which vary in accessibility, speed, flexibility, and security. Presented here is a brief survey of database tools and categories available for use by the Java™ platform.

3.1 Categories

3.1.1 Hash files

Hash files have the approximate functionality of a hash table; i.e one key field for searching and one containing the actual data. The interface is usually trivial, often based on java.util.Hashtable which is part of the standard Java™ class library.

This category of databases seldom incorporate the concurrency control necessary for multiuser access. Client/server access is also very uncommon. RMI\(^1\) and CORBA\(^2\) are named by a few projects as the solution to these issues.

The fact that these databases operate on only a few local files is appealing to many who develop small scale applications that require persistence and very moderate searching capabilities. These applications are typically easy for end-users to install as they require no database accounts to be set up.

\(^1\)http://java.sun.com/products/jdk/rmi/
\(^2\)http://www.corba.org/
3.1.2 Relational Databases

Relational databases are constructed as a collection of tables with relations. Each table column represents a field; one or more fields can make up a key which can take part in a relation. All tables have a designated primary key on which searches are the quickest.

<table>
<thead>
<tr>
<th>CPU</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>CLOCK</td>
<td>PRICE</td>
</tr>
<tr>
<td>Dyron</td>
<td>600</td>
<td>58.95</td>
</tr>
<tr>
<td>Dyron</td>
<td>650</td>
<td>64.95</td>
</tr>
<tr>
<td>Yelleron</td>
<td>600</td>
<td>89.95</td>
</tr>
<tr>
<td>Yelleron</td>
<td>633</td>
<td>94.95</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MANUFACTURER</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>FULLNAME</td>
</tr>
<tr>
<td>DMA</td>
<td>Device Manufacturing Association</td>
</tr>
<tr>
<td>UNTIL</td>
<td>Until Corporation</td>
</tr>
</tbody>
</table>

Example 3: Relational database.

A key is not a key unless it is guaranteed to be unique. This property is one of many constraints that are supported by the relational paradigm. In example 3, both \textit{NAME} and \textit{CLOCK} are necessary to form a unique key for \textit{CPU}; \textit{MANUFACTURER} does only need its \textit{NAME} field to form a key. The relation between the two tables are the \textit{CPU.MANUF}→\textit{MANUFACTURER.NAME}.

The JDBC™ interface allows Java™ access to a wide range of relational databases through explicit use of the SQL, the query language standard of the relational paradigm. Example 4 on the next page shows how to retrieve information using the JDBC™.

3.1.3 Object Databases

Object oriented database management systems, OODBMS, were developed to meet the needs of object oriented programming languages, thus much of the design was borrowed from OOPL research.

Each object in an OODBMS has a unique and immutable object identifier, OID, which is used internally for inter-object references [EN00]. This allows two objects to be equal albeit not identical, object identity is always preserved.

An object definition, or class, can be derived from another using inheritance. This provides the user with an incremental way of defining classes. Object databases also allow complex data structures, often with collection types such as lists and sets; relational databases requires you to create new tables in which keys are listed.

Object behavior, as described in [EN00], is based on the operations available for the object. Hidden attributes coupled with retrieval methods invocable on the DBMS provide

// Make a connection using driver, name, and password.
Connection con = DriverManager.getConnection("jdbc:myDriver", 
   "myLogin", "myPassword");

// Create an SQL statement which will join the two tables.
Statement stmt = con.createStatement();
String sql = "select CPU.NAME, CLOCK, PRICE, FULLNAME 
   from CPU,MANUFACTURER 
   where MANUF=MANUFACTURER.NAME;";

// Query database and perform operations on results.
ResultSet rs = stmt.executeQuery(sql);
while (rs.next()) {
   String price = rs.getString("PRICE");
   // ...
}

Example 4: Using JDBC™.

encapsulation, a key concept of the OO paradigm. This encapsulation is however often broken [SC96] by ad hoc queries since they require arbitrary computations on the data, which is why substituting the encapsulation on the database level with a security layer has been discussed in [Dat95].

Many object databases have native Java™ bindings and can thus be seamlessly integrated. Instead of SQL-based queries, data management and retrieval is currently converging towards the ODMG 3.0\(^4\) standard put forward by the Object database management group. Queries are formed using the object query language, OQL, which is part of the ODMG.

The ODMG provides a uniform data model, the lack of which has historically been the one of the biggest problems with OODBMS [DD95]. This data model is sufficiently similar to that of Java™ to solve the impedance mismatch [SC96] [ABD+89] problem of relational database systems; one data model for Java™ and one for the query language (i.e SQL) means translation overhead and potentially loss of information.

3.1.4 Object-Relational Databases

Object-relational database management systems include support for complex types and inheritance which are not present in databases belonging to the pure relational paradigm. This means that the impedance mismatch problem is typically reduced when moving to an ORDBMS.

Instead of using the ODMG, a new extended version of SQL is used [EM99]. From a

\(^4\)http://www.odmg.org/
Java™ point of view, this means using the JDBC™. The new model is backwards compatible, which means that legacy systems can remain in operation while the organization moves to an object oriented solution.

3.2 Tools

3.2.1 PPHash

PPHash is a persistent hash table package containing the PPHash class. It implements the same methods as the java.util.Hashtable class but keeps the data in files that can be accessed later [Gre00].

The database is uncached and creates a separate file for each object saved, making the performance highly dependent upon the underlying file system.

```java
// Saves files in directory "db".
PPHash hash = New PPHash("db");

// Any serializable object can be used for keys and values.
hash.put("somekey", new Double(12345.0));
```

Example 5: Using PPHash.

The software is available for free download, being licensed under the GNU GPL ⁵.

3.2.2 XL2

An XL2 database is a network of serialized objects. It uses persistent references to scale Java™ object serialization [Han00]. For an explanation of serialization, see section 4.1.1 on page 13.

This database, like the previous, belongs to the hash file category. XL2 does however possess a number of features otherwise reserved for more advanced databases, such as basic transaction support (see example 6 on the next page).

XL2 also sports features like automatic garbage collection and compaction, keeping the database from growing in an uncontrolled fashion. A cache is also implemented to increase performance.

A root object is explicitly set for the database using setRoot, any object referenced by XL2References (as seen in example 7 on the facing page) is then made persistent.

⁵http://www.gnu.org/copyleft/gpl.html
// Saves files in directory "db".
XL2Database db = new XL2Database();
db.open("db", db.OPEN_READ_WRITE);

Transaction t = db.newTransaction();
t.begin();
// Perform operations.
t.commit();

---

Example 6: XL2 transactions.

---

public class Person implements java.io.Serializable {
    protected String name = null;
    protected XL2Reference mother = null;
    protected XL2Reference father = null;
    protected XL2Reference[] children = null;
    protected XL2Reference spouse = new XL2Reference();
    public Person(String name, Person mother, Person father) {
        this.name = name;
        this.mother = new XL2Reference(mother);
        this.father = new XL2Reference(father);
    }
    public void setSpouse(Person spouse) {
        this.spouse.set(spouse);
    }
    public Person getSpouse() {
        return (Person)(spouse.get());
    }
    // ...
}

---

Example 7: XL2 references.

---

This piece of Java™ software is also released under the GNU GPL.

3.2.3 MySQL

The relational database MySQL [MyS00] is a popular choice for developers of dynamic web pages, much thanks to its speed and ease of use. Enterprise use is however limited as there is no support for transactions in the standard distribution.
MySQL is based on a client/server architecture. JDBC™ drivers, such as MM MySQL⁶, allow access to servers from within Java™. Example 8 replaces the database connection code of example 4.

```java
// Load the driver. 
Class.forName("org.gjt.mm.mysql.Driver");

// Make a connection.
Connection con = DriverManager.getConnection("jdbc:mysql://localhost/test", "myLogin", "myPassword");
```

Example 8: Making a MySQL connection using JDBC™.

MySQL recently changed licence to GNU GPL.

### 3.2.4 ozone

Ozone [ozo00] is an open source object oriented database management system written in Java™. Besides having its own native API, ozone is mostly ODMG 3.0 compliant. Example 9 demonstrates ODMG style initialization, data retrieval and basic transaction usage. Sadly, OQL queries are not currently available.

An impressive extra is the fully W3c compliant DOM implementation which allows ozone to act as a repository for XML documents. XPath queries are implemented and an update language called XUpdate is in the works.

```java
// Open a database connection.
ODMG odmg = new OzoneODMG();
Database db = odmg.newDatabase();
db.open( "ozone:remote://localhost:3333", Database.OPEN_READ_WRITE );

// Retrieve a named object.
Object root = db.lookup("root");

// Perform a transaction.
Transaction tx = odmg.newTransaction();
tx.begin();
// ... 
tx.commit();
db.close();
```

Example 9: Using the ODMG interface of ozone.

⁶http://www.worldserver.com/mm.mysql/
The code is released under the Ozone Library License\(^7\).

### 3.2.5 PostgreSQL

PostgreSQL\(^{[Po00]}\) is an ACID\(^8\)-compliant object-relational database. ACID basically means that the database management system supports transactions and has logging capabilities that allows the database to remain consistent if, for example, the power should fail. This, combined with the availability of commercial support, makes PostgreSQL a commercially viable database solution.

A JDBC\(^{TM}\) driver is bundled with the source distribution and is used in the same fashion as MySQL, as example 10 demonstrates.

```java
// Load the driver.
Class.forName("org.postgresql.Driver");

// Make a connection.
Connection con = DriverManager.getConnection("jdbc:postgresql:test",
    "myLogin", "myPassword");
```

Example 10: Making a PostgreSQL connection using JDBC\(^{TM}\).

There is an ongoing effort to make the ODMG interface available for PostgreSQL\(^{9}\). This would make a separate mapping layer unnecessary.

PostgreSQL is free software.

---

\(^7\) [http://www.ozone-db.org/ozone7.html](http://www.ozone-db.org/ozone7.html)

\(^8\) Atomicity, Consistency, Isolation, and Durability

4 Mapping

This section makes a survey over available categories and tools of mapping between objects and databases.

4.1 Categories

There are a number of different approaches to providing databases with a comfortable view of objects. Most of these are directly connected to a specific kind of database, others span multiple paradigms.

4.1.1 Serializable

Java™ provides a streamable mapping for objects implementing the Serializable interface. All attributes not marked as static (common for all instances of a class) or transient (explicitly kept from being serialized) are stored.

When added to an ObjectOutputStream, both primitive values and references are converted into binary code. All referenced objects are themselves added to the stream in a recursive manner.

```java
public byte[] convertToBytes(Object obj) {
    ByteArrayOutputStream bs = new ByteArrayOutputStream();
    ObjectOutputStream os = new ObjectOutputStream(bs);
    os.writeObject(obj);
    os.flush();
    os.close();
    return bs.toByteArray();
}
```

Example 11: Serializable.

In example 11, the output stream is connected to a stream which produces a byte array of the serialized data.

Using an ObjectInputStream, the serialized data can later become real objects again anywhere the class bytecode is available. The data can cross both platforms and networks.

Java serialization has a number of drawbacks. The following is a quote from [KK00] where it is labeled critical:

```
Serialized objects does not survive versioning. If a library has changed and some serialized classes are not compatible anymore, the objects can not be loaded.
```
The serialized data is compact and well defined, but neither extensible nor human readable. The extensible markup language, XML\textsuperscript{10}, has by many been put forward as the solution to the problem. Two of the most elegant implementations to date are the Koala Bean Markup Language [KK00] and its sister project Koala Object Markup Language.

### 4.1.2 Manual Mapping

Another way of exporting information from objects is by manually mapping the data into something which is useful to the database, a process often referred to as marshalling.

Example 12 shows how this is done in JADE framework\textsuperscript{11} which among other things maps Java™ objects to XML. A list of fields (separated into attributes and elements for the sake of XML) is retrieved by calling the `getAttributes()` method and can later fed to the constructor to produce a copy of the original object.

Security measures can be added to this kind of marshalling to prevent access by other parts of the code than the persistence layer. The code shown in example would, if `value` was supposed to be kept an internal secret, break the encapsulation.

```java
public Real(Attributes attributes, Elements content) {
    value = attributes.getDouble("value");
}

public Attributes getAttributes() {
    Attributes attributes = new Attributes();
    attributes.add("value", value);
    return attributes;
}
```

Example 12: Manual mapping.

### 4.1.3 Introspection

Accessors, as shown in example 13, are often used in conjunction with introspection, a method of retrieving information about a class for which the interface was unknown at compile-time. The `java.lang.reflect` package holds classes which provide information about fields and methods; their types and modifiers. The reflection package can for example be used to find all public methods whose names begin with `get` or `set`, invoke them all and allow the results to be stored in a hash table with the remainder of the method names as keys.

Public fields, and with some code to relax security also the protected and private, can be exported using the same technique. Allowing direct access to fields is however not good object oriented practice.

\textsuperscript{10}http://www.w3.org/XML/
\textsuperscript{11}http://www.jade.dautelle.com/jade/
Example 13: Accessors.

The introspection approach requires that persistable objects have a default constructor declared, i.e. one without parameters.

4.2 Tools

4.2.1 Castor

Castor [Exo00] is a collection of advanced mapping frameworks which allows Java™ objects to be transformed to and from XML elements and relational database content. Castor also provides interfaces for LDAP directories and Enterprise Java™ Beans, which are two persistence solutions not covered by this paper.

To retrieve data from an object, Castor uses class descriptors, which contain enough information about a class to allow information to be extracted from its instances. These descriptors can be produced in any of the following ways:

Compile-time descriptors are either hand crafted or generated implementations of the ClassDescriptor interface. Each field has a handler which provides operation on its owner. When written as inner classes, the descriptors and handlers can provide access to private fields without otherwise compromising encapsulation.

Run-time descriptors have a default mapping, which is constructed using introspection on the accessors and public fields. Public fields are only made available if no accessors are present. The default mapping can either co-exist with or be replaced by a mapping file, which is expressed in XML.

The relational mapping framework allows locking, with dead-lock detection, and transactions with rollback. It also provides a configurable cache which utilizes the LRU algorithm

Castor’s OQL to SQL mapping allows for queries on objects in a relation database in much the same way as it would have been done in an OODBMS.

\[\text{See section 6.1 on page 21.}\]
Castor is subject to a BSD-like license\textsuperscript{13}.

### 4.2.2 JDO

JDO [Rus00] is an upcoming API from Sun Microsystems, which has at the time of writing undergone public review as part of the Java Community Process.

The purpose of the new API is to have a uniform persistence interface for Java™ independent of the underlying database. There are a great number of layers presently available, none of which have the same interface.

One of the main goals is the ability to switch JDO vendor with minimal changes to the code. Portability guidelines are provided to meet this end.

Just as Castor has different ways of making classes persistent, JDO provides two alternatives. Either a class explicitly implements \texttt{PersistenceCapable}, or the byte code is altered by a JDO enhancer to make it persistable. The enhancer requires an XML metadata file, much like the mapping file of Castor.

In the words of Castor developer Thomas Yip, Castor might eventually be “hammered” into JDO compliance. Different design decisions were made during the development of both products, but I expect at least a subset of the JDO interface will be available shortly after the final specification is released.

Currently, the only available implementation of the JDO is Transparent Persistence, which is part of Forte™ for Java™ Internet Edition\textsuperscript{14}.

### 4.2.3 jRelationalFramework

The jRelationalFramework [js00] is a lightweight framework proving object-to-relational mapping. The user is relieved of most of the JDBC™ work; a grasp of basic SQL concepts is however still required to use JRF.

Each persistent class is paired with a domain class; the word “Domain” is attached to the name of the first class. The domain class takes the role of metadata file and specifies the field names and their associated accessors as well as some SQL specifics. It typically also holds utility methods for oft-used SQL queries.

The framework makes exclusive use of accessors to access object data. Setters are obliged to mark the object as modified in order to make the database perform the update.

The product is released under the Mozilla Public License\textsuperscript{15}, as well as GPL and LGPL\textsuperscript{16}.

\textsuperscript{13}http://castor.exolab.org/license.txt
\textsuperscript{14}http://www.sun.com/forte/ffj/
\textsuperscript{15}http://www.mozilla.org/MPL/MPL-1.1.html
\textsuperscript{16}http://www.gnu.org/copyleft/lesser.html
5 Garbage Collection

As you are no doubt aware, Java™ automatically removes objects from memory when they are no longer reachable by references. This process is called garbage collection and relieves the user of having to explicitly allocate and free memory.

A program can be considered a cobweb of data, with objects tied together by numerous references. A few of the objects are attached to the ceiling, keeping the entire web from falling to the floor. The ceiling of this analogy is comprised of the root set of objects, which among others include the static reference variables of loaded classes and the references on the call stacks of the currently running threads.

Figure 1 to the left shows five objects, A through E. The first three are directly referenced by root objects and are therefore safe from garbage collection. D is given this immunity by object C which has a reference to it, thus providing a straight path of references up to the root. Object E, on the other hand, is not reachable via any other object and is therefore subject to garbage collection.

Note that simply being referenced does not make an object free to stay resident in memory, there must be an unbroken chain of strong references up to the root. The use of the word strong for common references will be made clear by the following paragraphs.

The introduction of reference objects in JDK 1.2 had a substantial impact on garbage collection, both in terms of control and of confusion. For this report it is sufficient to name two of the new reference classes, WeakReference and SoftReference. The former holds its reference only as long as there is a strong reference to its referent, whereas the other keeps its reference as long as there is sufficient spare memory. As we shall see later on, these two types of reference classes will lead to two different algorithms.

```java
Customer strong = new Customer("John Smith");
WeakReference weak = new WeakReference(strong);
SoftReference soft = new SoftReference(strong);
```

Example 14: Reference types.

Example 14 shows how weak and soft references are made. Reference objects are, as the name implies, simply objects containing references to other objects. What makes
them special is that the references they contain are exempt from the reference counting algorithm used by the garbage collector, meaning that they will not stop an object from being collected.

The objects marked A and B in figure 2 have paths of strong references to the root and is thus kept resident in memory. Object C is strongly referenced by A and remains as well.

![Figure 2: Weak reference active.](image1.png)

![Figure 3: Weak reference cleared.](image2.png)

If the strong reference is withdrawn and the object C is left with only the weak reference from B, the garbage collector will soon reclaim it. In figure 3, note that the reference object (marked R) now contains a null reference; the clearing of weak references is done automatically by the garbage collector. The reference object itself is however not removed as it is still strongly reachable by the root.

Example 15 holds annotated code which combines weak and strong references as well as introduces a problem associated with the programming of garbage collection hooks.

```java
String strong = new String("Data");
WeakReference weak = new WeakReference(strong);

System.out.println(weak.get()); // Prints "Data".
strong = null; // Removes the strong reference.
System.out.println(weak.get()); // Prints "Data" or "null".
```

Example 15: Resetting references.

When the last strong reference to an object is cleared, weak references will eventually be cleared as well. After they are cleared there will another brief period of time before the object is actually removed from memory. In the example above, there is no way of knowing if the data is still weakly reachable.

Making code depend on the timing of various stages of the garbage collection process
is generally regarded as bad practice as no guarantees are made with regard to time, and implementations differ between virtual machines.

The last important detail on garbage collection as far as this paper is concerned is also the last\(^{17}\) step of the garbage collection process. Just before they are physically removed from memory, objects get a chance to do last-minute cleanups. If declared, the method `finalize()` is called. Example 16 provides a brief demonstration.

```java
protected void finalize() {
    System.out.println("Good bye cruel world!");
}
```

Example 16: Using a finalizer.

For more information, read [Paw00].

\(^{17}\)Unless the object makes itself reachable again using this mechanism.
6  Paging Theory

6.1  Demand Paging and Page Replacement

The concept of demand paging is a well known topic in database and operating system research. When adopted by a virtual memory system, this method allows individual pages of memory to be retrieved from disk instead of swapping the entire process to and fro. A page in this context is a block of memory.

When a page is accessed that is residing on disk, a page fault occurs and the page is retrieved. As each page fault results in time consuming I/O we want to minimize their occurrences. If we were able to keep every accessed page in memory then we would not have a problem, but reality requires us to eventually send some pages back to disk.

Page replacement algorithms have over the years been devised to handle this problem. I will briefly outline a few of them.

The Optimal Algorithm  dictates that we replace the page that will not be used for the longest period of time. This is purely theoretical as it requires information about the future.

The LRU Algorithm  chooses the least recently used page. This is not only a fairly good approximation of the optimal algorithm, but also possible to implement as it only requires information about the past.

One implementation approach would be to keep a stack of the pages where accessed ones are moved to the top and the pages at the bottom are subjects for replacement.

The Second-Chance Algorithm  is a potentially less costly implementation than the above-mentioned LRU. A reference bit, which is set when its corresponding page is referenced, is checked when the queue of pages is traversed. Pages with their bit set are spared from replacement and are considered newly arrived. Reference bits are cleared during the traversal.

A circular queue implementation is preferable, as it will make keeping track of relative arrival time easier.

Demand paging is described in greater detail in [EN00]. Page replacement is covered by [SPG98].

6.2  Object Paging

In traditional paging as described in section 6.1, pages are equal-sized sequential blocks of raw memory. From inside Java™ however, memory addressing is not possible and thus neither is this kind of paging. Paging, in the context of this paper, is instead defined by an equality between pages and what I will refer to as umbrella objects.

Storing only a few bytes of data on disk is just as costly time wise as storing a full block. This fact translates into a low feasibility of sending discrete Strings and other
simple objects to disk. An umbrella object is a complex object having many aggregate objects, providing a large enough chunk of data to make disk access feasible.

Even if the underlying database is capable of handling small pieces of data as well as large ones, there is still an overhead associated with each persistable object. The granularity of objects is one of the many factors that need to be considered with each case design.
7 Method

The idea for this paper was conceived in a design phase for the Oneiro project\(^{18}\). Oneiro, being a world simulation software, is expected to have a near exponential growth in number of objects. Eventually allowing third-party additions to the software meant keeping the API as simple as possible and that would not include explicit database access. Inter-session persistence was also a main concern, as I wanted to be able to restart the Oneiro server without losing the data it had accumulated.

I wanted to keep the object graph intact to allow natural tree-walking instead of queries. This line of thinking brought me to reference objects. They would allow me to hold together the graph, and still allow objects to temporarily leave memory. When asked to return the reference they were keeping, the reference objects could retrieve their referent from disk if it was not present in memory.

The properties of the weaker references allowed me to use the garbage collector to choose which objects to store on disk, making consistency a non-problem as the objects were guaranteed to be current if they were in memory and otherwise the same would hold for the data on disk. Concurrent access to the data on disk would require exclusive locks and is therefore not recommended; concurrent access within a single JVM is however not subject to any limitations other than those imposed by threading in general.

To be able to store objects, I needed to find a suitable database with Java™ bindings and as orthogonal an object mapping as possible, meaning I wanted to be able to store any kind of attribute. This search became the survey part of the paper. As the above-mentioned Oneiro project is an Open Source effort, I concentrated on free software.

To make sure the object paging scheme would work in practice as well in theory, I decided to create a sample implementation with a minimum amount of bells and whistles.

\(^{18}\)http://www.acc.umu.se/~oneiro/
8 Results

8.1 System Overview

The sample implementation, which is included as appendix A, is described by the UML diagram of figure 4 below.

Persistable is the base class for all umbrella classes. Any extending class will be saved to disk during the garbage collection process, courtesy of a database access embedded in the finalize() method. For an object to be available for collection, no strong references may be made to it. Instead we use the aggregate object Ref which holds a weak reference which can be retrieved when needed, using its get() method. Let me emphasize this, as it is very important:

No strong reference may be kept to a persistent object, other than locally within a method.

The exception to the above rule would be a list in which references are added and removed according to some page replacement algorithm, such as those described in section 6.1. This feature, which would minimize unnecessary disk I/O, is not available in the sample implementation but might prove crucial for real-world applications.

Replacing the WeakReference of Persistable.Ref with a SoftReference is an easy way of adding caching capability similar to the ones achieved by writing explicit page replacement code, as the soft references themselves usually implement a version of the LRU algorithm.

---

19 As introduced in section 6.2.
Figure 5 depicts the tree of a small World. In the provided test program, a World consists of many persistable Nodes. The tree can be walked and at each node a short piece of text can be written and later read. When a node is left, no strong reference is kept to it and it will eventually be paged to disk.

The database chosen for the sample implementation is a JNI wrapper library for the GNU GDBM hash file. The library is called JDBM and is written by Gunther Schadow. JDBM was chosen because it did not use multiple threads or a client/server architecture; either of which would have added to the timing problems being battled.

As JDBM uses serialization for mapping, all attributes that are neither static nor transient are saved with the object when the finalize() method of Persistable is called by the garbage collector. Unless the object pointed to by a Persistable.Ref is in memory, the get() method will reconstruct the object using data from the database.

The hash table key used is a unique object id which is constructed using the trivial MAX+1 algorithm, which states that a new object will receive an OID of one more than the currently highest number.

8.2 User Guide

The source code is available as appendix A and can also be found on the author’s homepage, http://www.acc.umu.se/~mortis/.

Make sure that you have installed JDBM and that jdbm.jar is in your CLASSPATH. Example 17 shows how the sample program is started. User input will appear underlined throughout this guide.

20 http://aurora.rg.iupui.edu/~schadow/dbm-java/
Welcome to the root node!

0 - 99 | up | describe | quit >

Example 17: Starting the sample program.

When the program starts for the first time, a new world tree is generated. The example above shows 100 subnodes for the root node, numbered from 0 to 99. Example 18 shows how to walk up and down the tree.

0 - 99 | up | describe | quit > 10

This node has no description.
0 - 8 | up | describe | quit > u

Welcome to the root node!

Example 18: Walking the world tree.

As only the current node is accessed using strong references, nodes are almost instantly sent to the hash file upon creation. Any changes made to nodes are committed to secondary storage after a short period of time following a node switch. The only change possible in the test program is adding pieces of description, as seen in example 19.

0 - 99 | up | describe | quit > d
Description: This is a piece of information.

Welcome to the root node!
This is a piece of information.

Example 19: Adding to a node’s description.

Persistable objects are not only kept on disk to reduce memory usage, this implementation also allows persistent state between sessions. Note that when we restart the program in this last example, the description added for the root note is still present.
Welcome to the root node!
This is a piece of information.

Example 20: State persistence between sessions.

8.3 Pitfalls

Avoiding explicit strong references to Persistable objects is much easier than the implicit ones. Take for example the code below, in which class A has an inner class B which provides class C with callback access. B has an implicit strong reference A.this which points to its parent, keeping it resident as long the callback object is strongly reachable.

class A extends Persistable {
    class B implements Callback {
        // [ Callback methods ]
    }

    B getCallback() { return new B(); }
}

class C {
    Callback cb;

    void assignQ(A.Ref r) {
        cb = ((A)r.get()).getCallback();
    }
}

Example 21: The dangers of non-static inner classes.

Either make sure that instances of non-static inner classes are referenced using WeakReferences, or adjust the code in a way similar to example 22.
class A extends Persistable {
    static class B implements Callback {
        A.Ref ref;

        B(A.Ref ref) { this.ref = ref; }

        // [ Callback methods ]
    }

    B getCallback() { return new B(ref()); }
}

Example 22: Using static inner classes.

There are number of other ways the paging scheme can be incapacitated, most of which can be eliminated using common sense.

8.4 Limitations

Since the test program uses a constant number of nodes, no method for permanently removing objects was implemented. All Persistables are kept in the database, regardless of reachability.

An easily remedied limitation of the sample implementation, is the maximum number of creatable Persistables which is set at Integer.MAX_VALUE.

8.5 Problems

The main problems encountered during the development of the sample implementation were of the race condition variety. This was not at all unexpected as the garbage collection process is stochastic. When a weak reference is cleared by the garbage collector, there is a brief period of time before finalize() is called. During this time there is no way to reach the object as it has not yet been saved to disk and all references to it in memory are lost. Any accesses to the object will for this reason have to be momentarily blocked.
9 Discussion

9.1 Implementation

The implementation very effectively masquerades the database access while still retaining the structure of the object graph. It also provides adequate inter-session persistence. More importantly, the software has shown that a stable solution is possible to develop.

9.2 Persistence Solutions

The paging scheme is a good alternative for maintaining large numbers of objects forming complex graphs. Cooperative development is made easy by abstracting the specifics of database access.

Compatibility with legacy relational database software can be provided using the mapping tools available. Concurrent access to objects is however not possible, except within a single JVM. Using a relational database when designing a new system, however, will likely lead to more overhead than is justifiable.

Object databases usually provide a good Java™ interface with minimal overhead for conversion between data models. The querying capabilities may be less than when dealing with RDBMSs, but the paging scheme will typically need nothing more than OID lookup.

Object-relational databases can replace object databases in situations where an organization has standardized on an ORDBMS and is not willing to make a new investment. This works especially well if the database comes with an ODMG compatible API.

Hash files almost never come with transaction support and are therefore not good for mission-critical applications. They usually don’t scale very well either, leaving out enterprise level use. The ease of use and speed when used as part of smaller projects does, however, make them a valuable alternative.

Traditional access to databases is still to be preferred in most applications, as the paging scheme does not work well with concurrent access. When transactional correctness is important, developers should also go with the traditional approach; object paging only allows transactions on a per umbrella object basis, which may leave the system in an inconsistent state should it fail and have to be restarted.

9.3 Object Paging vs Demand Paging

Demand paging, as used in virtual memory systems, handles actual blocks of memory. The page replacement algorithms perform exactly the task that the name implies. When used in the context of object paging, what is replaced is merely references; the memory blocks are controlled entirely by the garbage collector.

The implementation provided in appendix A is not fitted with any replacement code, which means almost instant removal of objects from memory upon dropping of strong references. As there is no caching of objects, unnecessary disk access is to be expected for objects that are used often.
9.4 Future

Among the topics not covered in this paper are Enterprise Java Beans™ and session bean persistence, LDAP directories as an alternative to relational databases, and persistence schemes such as checkpointing.

Further research regarding the efficiency of the paging scheme, using different page replacement algorithms for different purposes, would be of interest.

A list of important considerations to be made when adopting the scheme to a particular system could be written, as well as a more detailed analysis on where it is at all appropriate.
10 Acknowledgments

I would like to take this opportunity to thank some of the people who have made this paper possible:

Gunnar Gunnarson, whose many hours spent on assuring the quality of this paper has been greatly appreciated.

Ola Ågren, who was a big help in my quest for relevant articles.

Johanna Högberg, Sara Vestman, and the rest of the folks at Roaming Factory for making me feel welcome. A special thank you goes to Malin Sjöström for always keeping a fresh supply of cookies handy.

My fiancée Anna Burholm, for her love and support. Without her I would never have made it this far.
References


[Rus00] Craig Russell. JSR 000012: Java Data Objects (JDO), version 0.8, July 2000.

import java.lang.ref.*;
import java.util.*;
import jdbm.*;
import java.io.*;

/**
 * The base class for all persistable objects, i.e the umbrella objects.
 * @author Markus Mårtensson
 */
public abstract class Persistable implements Serializable {

/**
 * The associated reference object.
 */
private transient Ref reference;

/**
 * The shared database connection.
 */
private static Database database;

/**
 * The counter for the MAX+1 algorithm.
 */
private static int nextid = 0;

/* Establish access to the database. */
static {
    try {
        database = new Database(new File("objects.db"),
            Database.O_CREATE);
    } catch (IOException e) {
        throw new FatalException(e);
    }

    try {
        String n = database.fetch("nextid");
        nextid = Integer.parseInt(n);
    } catch (NoSuchKeyException e) {
        // Fallthrough, nextid stays at 0
    }
}

/**
 * Creates a new OID and sets up a reference object for new
 * persistable objects.
 */
protected Persistable() {
    synchronized(database) {
        nextid++;
        System.err.println("Creating [+nextid="]);
        reference = new Ref(this, nextid);
    }
}
/**
 * Returns the reference object associated with this object.
 * @return the associated Persistable.Ref.
 */
public Ref ref() {
    return reference;
}

/**
 * Retrieves a reference to the root object.
 * @return the root object.
 * @throws RootNotFoundException if no root is created.
 */
public static Ref root() throws RootNotFoundException {
    if (nextid > 0) {
        Ref r = new Ref(null, 1);
        r.locked = false;
        // Make sure the reference is unique.
        r = (Ref)r.readResolve();
        return r;
    } else {
        throw new RootNotFoundException();
    }
}

/**
 * Called before the object leaves memory. Converts the object into
 * a byte stream which is then stored in the database.
 */
protected void finalize() throws Throwable {
    try {
        ByteArrayOutputStream bs = new ByteArrayOutputStream();
        ObjectOutputStream os = new ObjectOutputStream(bs);
        os.writeObject(this);
        os.flush();
        os.close();

        database.store(String.valueOf(reference.id),
                          bs.toByteArray(), Database.S_REPLACE);
        database.store("nextid",
                          String.valueOf(nextid), Database.S_REPLACE);
    } catch (IOException e) {
        throw new FatalException(e);
    } catch (KeyExistsException e) {
        throw new FatalException(e);
    } synchronized(reference) {
        reference.notifyAll();
    }
}
public static class RootNotFoundException extends Exception {

    private Reference reference;

    public RootNotFoundException(Reference reference) {
        this.reference = reference;
        reference.locked = false;
    }

    public static class RootNotFoundException extends Exception {
        private Reference reference;
        private boolean locked;
        public RootNotFoundException(Reference reference) {
            this.reference = reference;
            reference.locked = false;
        }
    }

    public static class FatalException extends RuntimeException {
        private static final int serialVersionUID = 0;
        private Throwable throwable = this;
        public FatalException(Throwable throwable) {
            this.throwable = throwable;
        }
        public String getMessage() {
            return throwable.toString();
        }
    }

    public static class Ref implements Serializable {
        private static final long serialVersionUID = 0;
        private int id = -1;
        private WeakReference weak;
        private synchronized boolean locked = true;
        private static List refs = new LinkedList();
        public Ref(Object referent, int id) {
            this.id = id;
            weak = new WeakReference(referent);
            synchronized(refs) {
                refs.add(new WeakReference(this));
            }
        }
        protected int getId() {
            return id;
        }
        protected WeakReference getWeak() {
            return weak;
        }
    }
}
/**
   * Retrieves a strong reference to the referent. If it is
   * not in memory, the object is reconstructed using the
   * byte stream stored in the database.
   * @return the referent.
   */
public Persistable get() {
  Object o = weak==null ? null:weak.get();

  // The object is not resident in memory. Retrieving.
  if (o==null) {
    System.err.println("Loading ["+id+"].");
    
    /*
     * For a brief period of time, objects are neither reachable
     * via references nor available on disk. We have to wait.
     */
    while (locked) {
      synchronized(this) {
        try {
          wait(100);
        } catch(InterruptedException e) {
          throw new FatalException(e);
        }
      }
    }
    System.err.println("Waiting done for ["+id+"].");
    locked = true;

    // Retrieve and reconstruct.
    try {
      byte[] data = database.fetchBytes(String.valueOf(id));
      ByteArrayInputStream bis = new ByteArrayInputStream(data);
      ObjectInputStream ois = new ObjectInputStream(bis);
      o = ois.readObject();
    } catch (IOException e) {
      throw new FatalException(e);
    } catch (ClassNotFoundException e) {
      throw new FatalException(e);
    } catch (NoSuchKeyException e) {
      throw new FatalException(e);
    }
    weak = new WeakReference(o);
    ((Persistable)o).reference = this;
    System.err.println("Loading done for ["+id+"].");
  }
  
  return (Persistable)o;
}
/**
* Returns a hash code value for the object. In this implementation,
* it happens to be the OID.
*  *
* @return a hash code value for this object.
*/
public final int hashCode() {
    return id;
}

/**
* Indicates whether some other object is "equal to" this one.
* This implementation requires the most discriminating possible
* equivalence relation on objects; x and y are only equal if
* x==y.
*  *
* @param obj the reference object with which to compare.
*  *
* @return true if this object is the same as the obj
* argument; false otherwise.
*/
public final boolean equals(Object other) {
    return this==other;
}

protected void finalize() {
    System.err.println("Reclaimed Ref ["+id+"]");
}

/**
* This method makes sure there are only unique reference
* objects in memory. This method is called by the serialize
* marshaller.
*  *
* @return this object, or another reference object already
* in memory which handles the same OID.
*/
private Object readResolve() {
    System.err.println("readResolve for ["+id+"]");

    synchronized(refs) {
        for (Iterator i = refs.iterator(); i.hasNext(); ) {
            Object o = ((WeakReference)i.next()).get();
            if (o == null) {
                i.remove();
            } else if (((Ref)o).id == id) {
                System.err.println("Found an in-memory copy.");
                return o;
            }
        }
    }
    return this;
}
A.2 World.java

```java
import java.util.*;
import java.io.*;

/**
 * The main class for the test program. Contains a walkable tree of
 * Nodes.
 * @author Markus Mårtensson
 */
public class World {
    /** The root node. */
    private Node.Ref root;
    /** The node currently visited during the tree walk. */
    private Node.Ref current;
    /** A random number generator. */
    private Random random = new Random();
    /**
     * Creates, or loads, all the nodes of the world.
     */
    public World() {
        try {
            // If a root node is declared in the database, use that tree.
            root = (Node.Ref)Node.root();
        } catch (Persistable.RootNotFoundException e) {
            // If none exists, we create a new tree.
            Node node = new Node();
            node.addDescription("Welcome to the root node!");
            root = node.ref();
            fork(root, 100, 5);
        }
    }
    /**
     * Recursively creates subtrees for the World tree with a random of
     * number of subnodes for each node, the average of which is decreased
     * with depth.
     * @param subroot the root of the new subtree to be created.
     * @param num the maximum number of subnodes to create.
     * @param div divides <code>num</code> for the next level.
     */
    public void fork(Node.Ref subroot, int num, int div) {
        if (num>0) {
            int r = random.nextInt(num)+1;
            num /= div;
```
for (int i=0; i<r; i++) {
    Node.Ref n = new Node(subroot).ref();
    fork(n, num, div);
}

/**
 * Creates a new world and starts the user interface.
 * @param args parameters from command line. Ignored.
 */
public static void main(String[] args) {
    Runtime.runFinalizersOnExit(true);
    new World().play();
}

/**
 * Starts the console user interface of the test program.
 */
public void play() {
    current = root;
    PrintStream out = System.out;
    BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in));

    try {
        char cmd = ' ';
        do {
            Node node = (Node)(current.get());
            int num = node.numContents();

            out.println();
            out.println(node.toString());
            // Add node numbers to prompt, unless we have a leaf.
            if (num>0) {
                out.print("0 - "+(num-1)+" | ");
            }
            out.print("up | describe | quit > ");
            String reply = in.readLine();
            if (reply==null) {
                // EOF means quit as well.
                break;
            }
        } while (true);
    } catch (IOException e) {
    }
}
try {
    // If we have a number, find a matching subnode
    int sel = Integer.parseInt(reply);
    try {
        current = node.getContent(sel);
    } catch (IndexOutOfBoundsException e) {
        out.println("Invalid subnode number!");
    }
} catch (NumberFormatException e) {
    // The user input was not a number, check for commands.
    cmd = Character.toLowerCase(reply.charAt(0));
    switch (cmd) {
        case 'u' :
            current = node.getParent();
            if (current == null) {
                current = root;
                out.println("Cannot move beyond root.");
            }
            break;
        case 'd' :
            out.print("Description: ");
            String desc = in.readLine();
            node.addDescription(desc);
            break;
    }
} while (cmd != 'q');
} catch (IOException e) {
    // Fallthrough
}
}

A.3 Node.java

import java.util.*;

/**
 * A node contains a short text description of itself and references
 * to its own subnodes.
 *
 * @author Markus Mårtensson
 */
public class Node extends Persistable {
    /** The description of the node. */
    private StringBuffer description = null;

    /** The parent of the node. */
    private Ref parent = null;

    /** The subnodes of the node. */
private List contents = null;

/**
 * Creates the root node of the world.
 */
public Node() {
    super();
}

/**
 * Creates a subnode.
 * @param parent the parent of the new node.
 */
public Node(Ref parent) {
    this();
    Node r = (Node)parent.get();
    // Add this node to the parent.
    if (r.contents == null) {
        r.contents = new LinkedList();
    }
    r.contents.add(ref());
    this.parent = parent;
}

/**
 * Returns a string representation of the object. For nodes, this
 * means the textual description.
 * @return the description of the node.
 */
public String toString() {
    if (description == null) {
        return "This node has no description.";
    } else {
        return description.toString();
    }
}

/**
 * Adds a line of text to the description of the node.
 * @param desc a line of text to be added.
 */
public void addDescription(String desc) {
    if (description == null) {
        description = new StringBuffer();
    }
    description.append(desc);
    description.append("\n");
}
/**
 * Returns the number of subnodes in this node.
 * @return the number of subnodes.
 */
public int numContents() {
    return contents == null ? 0 : contents.size();
}

/**
 * Gets the subnode of this node having the given index number.
 * @param num the index number of the subnode.
 * @return the requested node.
 * @throws IndexOutOfBoundsException if <code>num</code> is invalid.
 */
public Ref getContent(int num) {
    return (Ref)contents.get(num);
}

/**
 * Gets the parent node.
 * @return the parent node.
 */
public Ref getParent() {
    return parent;
}

All the included code is subject to the following license:

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. Permission to use, modify, and redistribute the code is hereby granted.