Comparing the Performance of Object Databases and ORM Tools

PIETER VAN ZYL,
DERRICK G. KOURIE
AND
ANDREW BOAKE
ESPRESSO Research Group
Department of Computer Science
University of Pretoria

The currently popular distributed, n-tiered, object-oriented application architecture provokes many design debates. Designs of such applications are often divided into logical 'tiers' – usually user interface, business logic and domain object, or data, tiers, each with their own design issues. In particular, the latter contains data that needs to be stored and retrieved from permanent storage. Decisions need to be made as to the most appropriate way of doing this – the choices are usually whether to use an object database, to communicate directly with a relational database, or to use object-relational mapping (ORM) tools to allow objects to be translated to and from relational form.

Most often, depending on the perceived profile of the application, architects make these decisions using rules of thumb derived from particular experience or the design patterns literature. Examples include: object-oriented databases ease programming, relational databases ease report generation and data mining; object-oriented databases are good for navigation around an object model, relational databases are good for sequential processing and complex queries; if you are writing an application from scratch, use an object database, if you need to integrate to various sources of legacy data, use an ORM tool. Although helpful, these rules are often highly context-dependent and are often misapplied.

Research into the nature and magnitude of 'design forces' in this area has resulted in a series of benchmarks, intended to allow architects to more clearly understand the implications of design decisions concerning object persistence. In this paper, the performance of selected open source object persistence tools is investigated, to attempt to clarify the myths surrounding the performance of the different options. In particular, we compare Hibernate, representative of the ORM stable, and db4o, representative of object-oriented databases. The OO7 benchmark is used to compare the speed of execution of a suite of typical persistence-related operations in both candidates. We then propose some preliminary explanations of the sometimes surprising results.

---

1. INTRODUCTION

Many contemporary applications use data that needs to be stored and retrieved. In the object oriented environment, objects are usually used to represent data, and in this context, it is therefore *objects* that need to be persisted. Persistence implies a '*process* of how to store the objects' [Ambler 1998] as well as a persistence mechanism. There are three well-known classes of object persistence mechanism: Object Oriented Database Management Systems (ODBMS's) , Relational Database Management Systems (RDBMS's) and Object-Relational Database Management Systems. (Another of course, is the simpler mechanism of writing serialized object representations to a file.) Cattell et al [2000] define an ODBMS to be a DBMS that integrates database capabilities with object-oriented programming language capabilities. In such an ODBMS, both object attributes and object methods are stored in the database.

A well-known problem in persisting objects to a relational database is the so-called *impedance mismatch* that arises between both the object model and the relational model and between the object programming language and the relational query language [Cattell 1991]. To resolve the impedance mismatch problem, various hybrid solutions have been proposed. Thus, object-relational databases have been developed, and traditional RDBMS vendors have included object persistence capabilities into their products. Object Relational Mapping (ORM or also known as O-R) tools have also been developed in an attempt to bridge this mismatch and to make persistence of objects easier for the developer. These tools provide a mapping between the object model and the relational model, acting as an intermediary between an object oriented code base, and a relational database.

Author Addresses:
P. van Zyl, ESPRESSO Research Group (http://espresso.cs.up.ac.za) , Department of Computer Science, University of Pretoria, Pretoria, 0002, South Africa; pvzyl@csir.co.za.
A. Boake, Espresso Research Group (http://espresso.cs.up.ac.za) , Systemic Logic; andrew@systemiclogic.com
D. Kourie, Espresso Research Group (http://espresso.cs.up.ac.za) , Department of Computer Science, University of Pretoria, Pretoria, 0002, South Africa; dkourie@cs.up.ac.za

In choosing between these options for a particular application, one needs to consider various 'forces'. Chief among these are ease of development, performance and access to different sources of data, perhaps by more than just the application being developed. The different solutions resolve these forces differently. For instance, object databases ease development considerably by automating virtually all aspects of persistence, thus freeing the developer to concentrate on more pressing business modelling aspects. However, object databases generally hide their contents behind object oriented programming environments, which considerably complicates extraction of the data by popular reporting and data mining tools.

Another example concerns performance. Popular rules of thumb assert that an application that navigates the data (following links) will be more suited to object database use, while one that processes data items sequentially (for example, adding interest to clients' account balances), or one that performs complex queries ("Show me this client's total credit exposure") are more suited to relational databases. Popular belief also asserts that object-relational tools add translation overhead to all persistence operations, and so are proportionally slower that either of the other two solutions. But, they do allow developers to use more powerful object oriented domain modelling techniques, whose constructs are then translated to relational equivalents by the tool. But, specifying these translation rules again adds time and energy to the development process.

We have seen that the correct decision for a particular application is both context dependent and requires a good understanding of the relative importance, and of the actual magnitude, of the forces in the chosen solution. In our research, we are delving into more objective clarification of these forces. Essentially, we are asking the same questions that have been asked by countless software architects: Which is "better": the hybrid solutions or the pure solutions? Why are the pure solutions not more extensively used? What does "better" mean? And how do we know what is better for *this* application? But, going further, we intend to *measure* the contending forces and *understand* them.

In this paper, we report on a particular piece of research into the performance aspects of these choices. We use a benchmark to compare ODBMS's to ORM tools, with particular emphasis on the performance aspects. An object persistence benchmark compares solutions by subjecting them to a suite of representative operations. The particular benchmark we have chosen is OO7, widely used to comprehensively test object persistence performance. We have used its measurements to compare typical solutions that are available today in the pure and in the hybrid world: db4o, representative of object databases, and Hibernate, representative of the ORM stable. Both of these are popular Open Source products.

Because of its general popularity, reflected by the fact that most of the large persistence mechanism providers provide persistence for Java objects, it was decided to use Java objects for our studies. A consequence of this decision is that the OO7 Benchmark, currently available in C++, has had to be re-implemented in Java. The results reported here are based on a partial re-implementation of the benchmark. Work is currently underway to develop a fuller implementation, but the results reported here are necessarily limited in scope to re-implemented portions of the benchmark. This work is therefore a first step in investigating object persistence performance. Nonetheless, we believe it has yielded information of interest.

In the nineties, when ODBMSs were still rather new, there were a variety of studies to assess their performance. For example, Cattell and Skeen [1992] investigated the performance of various RDBMSs against ODBMSs, while Carey et al [1993] compared the performance of various ODBMSs. More recently, Jordan [2004] has made a study of all of Sun's persistence mechanisms. It was this latter study that inspired the idea of comparing ORM tools against ODBMS's.

In the next section, the OO7 Benchmark and the reasons for selecting it will be discussed. Section 3 explains the design constraints in implementing the Java version of the benchmark. Section 4 discusses the Java version in more detail. Our measurements are assessed in Section 5. Section 6 offers our conclusions and perceptions of future work to be done.

2.   BENCHMARKS

Questions about the performance of ODBMS's arose during the late 80's and early 90's, soon after academic and commercial ODBMSs were becoming available. It was at this time that several benchmarks were created to measure their performance. Well known benchmarks included HyperModel [Anderson et al 1990], OO1 [Cattell and Skeen 1992] and OO7 [Carey et al. 1993a]. The OO1 benchmark was intended to study the performance of engineering applications. The HyperModel approach was based on earlier versions of the OO1 benchmark. It incorporated a more complex model with more complex relationships and a wider variety of operations. The HyperModel benchmark focused on the hypertext model. The OO7 benchmark was based on both of these benchmarking efforts.

The next section will provide an overview of the OO7 benchmark and also discuss why this benchmark was selected for our benchmarking and comparison investigation.

2.1   OO7 benchmark overview
OO7 was designed to investigate various features of performance, and it included complex objects which were missing from the OO1 and HyperModel benchmarks [Carey et al 1993a]. While the earlier benchmarks used single value results, the OO7 benchmark provided a collection of results. Carey et al [1993a] indicate that the benchmark is intended to investigate 'associative operations, sparse vs dense traversals, updates to indexed vs. non-indexed object attributes', etc.

The benchmark uses a hierarchy of objects, modeling an engineering design library. Each Model contains either BaseAssembly or ComplexAssembly objects. ComplexAssembly objects contain other BaseAssembly objects. BaseAssembly objects contain CompositePart objects, that contain AtomicPart objects. AtomicPart objects are connected with each other through Connection objects. All of these objects have some basic attributes and some collections representing one-to-one, one-to-many and many-to-many relationships. Most of these relationships are bi-directional.

This object model is used as the target of various navigation and persistence operations. There are configuration settings that can be changed to influence the number of objects created, the number of models, the number of connections, etc. The benchmark also has a small, medium and a large database configuration which specifies the number of objects to be created in the benchmark. For each of these database configurations it is possible to configure the number of connections between objects. An example of this is where an AtomicPart is connected to other AtomicParts and with the use of the configuration value we can control the number of these connections [Carey et al 1993a]. Carey et al [1993a] defined 3, 6 and 9 as suggested connection values. In this way, we can have a small configuration with 3 connections, a small configuration with 6 connections, a medium configuration with 9 connections, etc.

The benchmark contains operations that operate on the design library and can be grouped into three categories: traversals, queries and modifications. Although these are well-documented in [Carey et al 1993b], various features of these operations that are not so obvious will be mentioned below. The modification operations refer to the insertion and deletion of objects, and to some of the traversal operations that not only traverse the object hierarchy but also modify the objects by swapping values, renaming them, etc.

Operations are run as *cold* or *hot* [Carey et al 1993a]. Cold runs are runs where all the caches should be empty, and hot runs are where caches are full. There is also the in-between notion of a warm run, which refers to an initial cold run to fill up the caches, followed by a hot run. Section 6 will provide the measurements in regard to these operations under both cold and hot running conditions.

One of the early difficulties that arose in attempting to use the OO7 benchmark was the issue of query languages [Carey et al 1994]. Not all of the systems tested then had query languages, and those that had, had languages that differed in capabilities. For those that did not have query language capabilities, C++ query methods were written in the hope of enabling comparison across all of the systems. Carey et al [1994] conceded that this could favour some implementations with no query language capabilities.

This possible bias is avoided in the present study, since both of the systems that we have chosen to investigate have their own object query facilities. Db4o uses SODA (Simple Object Database Access) as well as Native queries, and Hibernate uses HQL (Hibernate Query Language).

In selecting a benchmark for comparing our two chosen target systems, the OO7 was a natural prime candidate. Because it has a deep object hierarchy with many complex relationships (one-to-many and many-to-many), it is well-suited to assessing non-trivial object oriented applications. Furthermore, the fact that the code was available in C++ meant that it could be converted to Java with relative ease. Finally, we were influenced by the fact that it had been used in the recent past for research studies into persisting Java objects, for example [Jordan 2004].

3.   JAVA VERSION OF OO7 BENCHMARK

This section discusses the overall approach to producing a Java version of the OO7 Benchmark, the principle utility classes that were used, and the most important ways in which our version differs from the original C++ version.

3.1   Overall approach

The Java version was written in two basic stages. An initial attempt took the lead from another Java OO7 version that had been developed and distributed by the Ozone open source ODBMS [Ozone]. On investigation, it became apparent that Ozone's Java version was very basic. Not only was it not a full equivalent of the original C++ version, but valuable debugging and configuration settings available in the C++ version had not been incorporated into the Ozone version.

In a second stage, the original C++ code that had been used for the Versant implementation was taken as a starting point. An equivalent Java class was written for each C++ class. Here all of the language structures that were similar were copied over to the Java version. An example of this was where similar language structures (such as for-loops) were merely copied over from the C++ to the Java version. This was done in an effort to stay as close to the original version as possible. This easily led to a Java code equivalent of OO7, but which did not yet provide any persistence-related code. However, this was useful in and of itself, as it was deemed desirable to have a pure in-memory representation of the model. This was to serve as an absolute baseline in comparing cold and hot object operations across different platforms. The idea for doing this was taken from Jordan [2004]. Once the pure Java OO7 model was in place, the db4o and Hibernate implementations were produced. These will be discussed in Section 5.

3.2   Utility classes

A *Persistence* class was designed for use in both the db4o and Hibernate implementations. This class was used at all points in the benchmark model where persistence was needed. It has methods for saving, deleting, updating, etc, hiding the specific implementation details.

Each implementation then implemented its persistence code in a separate utility class whose methods were called by methods in the *Persistence* class. Followers of Design Patterns may recognize this as an example of using the Bridge pattern. The db4o utility class was called *Db4oUtil* and Hibernate's utility class, *HibernateUtil*. The intention was to enable us to hide the persistence mechanism used from the model. The idea was to use only one model, and to merely interchange the correct persistence mechanism classes when testing the different platforms.

This aspiration was not realized, as it was found that Hibernate required that the parent of a hierarchical object had to be saved before saving the children. This was not true for db4o. Thus, the order of calls to the persistence utility had to be different in the two implementations. This necessitated two separate implementations for the two platforms. The original C++ OO7 version also used this approach of writing separate implementations: one for Versant, one for Objectivity, etc.

3.3   Deviations from the benchmark

The original version of the OO7 Benchmark had configuration options for using one or many database transactions (i.e. database commit points). It was decided not to investigate the difference in using one or many transactions at this point. Db4o and Hibernate both provide transaction management. Db4o opens a transaction when the database is opened, so there will always be a transaction available. Hibernate provides transactions through the use of JTA (Java Transaction API) or JDBC transactions. No transactions were used in the Hibernate implementation. We mention this here in the interests of a more complete understanding of the basis for comparison.

It was also decided to exclude indexes for searching and queries, as we did not want to use any of the performance enhancements that are available to users of relational databases.

Additionally, the formula for computing the average hot times was changed by including the time take for the last run. Carey et al [1993b] had omitted the last run because they did want to include the overhead of commits. We argue that commits are an inherent part of the operation and they may have quite an important influence on the times. Indeed, in contrast to the original benchmark, instead of providing one big commit at the end, our version commits each time that an object is added or updated. Thus, the average hot time calculation has been changed to include all of the iteration times except for the first, this iteration time being the cold run.

Finally, the original OO7 version included so-called '*null* methods' to simulate work external to the database. These are called *doNothing*() methods in the code and are implemented as for-loops which simply request the system time. These were not implemented in the Java version, since there seemed to be no need for them. They are apparently intended to aid in simulating the overall time that would be taken for an application, and do not relate to the performance of the respective persistence mechanisms as such.

4.   PERSISTENCE MECHANISMS TESTED

This section discusses the two persistence mechanisms for which OO7 implementations were written. It also provides information about the testing environment.

4.1   TESTING ENVIRONMENT

The benchmark was run on a Fujitsu Siemens laptop with 1G of internal memory. The 80G hard drive has a drive speed of 4200rpm. The laptop has Mandriva Linux 2006 running with X. Although most applications were shut down while the benchmark was running, some applications to capture the data had to remain open. Furthermore, the laptop was not plugged into a network. As a result, there could be a certain amount of overhead in the measured times. However, there does not appear to be any reason why such overhead would influence one persistence mechanism more or less than the other. Nevertheless, as part of future work, the benchmark operations will be run on a dedicated machine with no other applications running.

4.2   OBDMS IMPLEMENTATION: DB4O

As previously pointed out, both the db4o and the Hibernate implementations to persist objects were derived from the Java OO7 code that simply generated objects in memory.

Often, adding persistence to your Java application requires rather intrusive additions and changes to your code. There are really three categories of changes required: telling the persistence mechanism *what* to persist, *how* to persist, and *when* to persist.

Telling the mechanism what to persist generally means marking the target classes and attributes. For example, one may be required to have your persistent objects implement an interface to mark them as persistent, or be part of a framework (for instance, be an Enterprise Java Bean). You may also need to code in a certain way so that the persistence mechanism can recognise patterns (for example getter and setter methods). Telling it how to persist depends a lot on the how the mechanism maps in-memory object representations to storage representations. For example, you may need to run the code through a pre-processor for persistence code to be added, or specify persistence-related information in configuration files. Telling a mechanism when to persist is a more difficult matter. In general, one is faced with the choice of storing every change to the objects in memory to the store (which is highly inefficient), or adopting some scheme which marks objects as 'dirty', to be stored later, when you tell it to (explicitly in the code).

Db4o is an open source object oriented database that easily and without extra work to tell it what and how to persist, stores Plain Old Java Objects (also known as POJOs). Db4o does however require us to tell it when to persist. In general, code to save or update objects in the persistence store was inserted just after the object had been created or changed in memory. In this way, producing the db4o implementation from the OO7 Java code was in fact quite straightforward.

When using db4o, one needs to keep track of the so-called *update* depth and *activation* depth [Db4o 2006]. Update depth has to do with the fact that when an object that has a reference to other objects is updated, db4o will only save the changes to referenced objects now if the update depth is greater than 1. This is used to control performance. The default update depth in db4o is 1. Since the benchmark makes use of objects that have sets of references to other objects and we wanted to save all of the changes immediately, this default parameter was changed.

Activation depth relates to retrieving objects from the database that reference other objects. Db4o can, if needed, retrieve the whole object graph referenced by the object being retrieved. This can be inefficient, especially if we don't need to use the entire object tree right away. So, when we retrieve an object that has references to other objects, we can defer retrieving the referenced objects until needed. This is done by a technique called lazy loading, where referenced objects are fetched only when a reference to them is followed. The tree depth at which this occurs is called activation depth in db4o. By default, db4o specifies activation depth as 5. In running the benchmark, the activation depth was increased to retrieve the whole referenced object graph.

Db4o can be run as an embedded database, as a local server in the same virtual machine (an embedded server) and as a remote server. For our implementation Db4o was run as an embedded database.

4.3   ORM TOOL IMPLEMENTATION: HIBERNATE

Hibernate is an ORM tool that stores in-memory objects to, and retrieves in-memory objects from, a relational database. To test Hibernate, we needed to change the in-memory version of the OO7 Java code to make use of Hibernate. Hibernate can be used with any relational database – we used Postgres for our implementation.

To more clearly understand the explanation of how this was done, we need to define some of the concepts. Ambler [2006] provides the following definitions:
—   "Mapping: The act of determining how objects and their relationships are persisted in permanent data storage, in this case a relational database.
—   Property: A data attribute, either implemented as a physical attribute, such as *String firstName,* or as a virtual attribute implemented via an operation, such as *Currency getTotal ()*.
—   Property mapping: A mapping that describes how to persist an objects property.
—   Relationship mapping: A mapping that describes how to persist a relationship between two or more objects (generally, association, aggregation, or composition).
—   Inheritance mapping: Mapping the inheritance hierarchy to relational database tables."

In using Hibernate, these mappings can be specified in xml mapping files or they can be defined in the Java code by using Xdoclet tags or annotations. We used Hibernate Xdoclet tags in the code, and extracted the tags from the Java files, using an ant task, to create the xml mapping files.

The mapping of the properties was relatively straightforward. Most of the time was spent on getting the relationship and inheritance mappings to work correctly. These are discussed in the following sections.

In the previous section it was mentioned that, when using db4o, one can set the activation depth of the objects retrieved. It is interesting to note that Hibernate also provides lazy loading, where it is possible to limit the number of objects returned.

4.3.1   Relationship mappings

Recall that relationship mapping is the mapping of one-to-one, one-to-many and many-to-many relationships between objects to their chosen relational database representations. Most of the types of object oriented relationship, including aggregation (is-part-of), are found in the OO7 benchmark. It was therefore necessary to specify how to handle these in our Hibernate implementation. This was found to be one of the most difficult parts of the implementation.

All the associations in the OO7 model are bi-directional, and so one has to consider how to specify relational queries to follow relationship navigation from both ends.

The one-to-many relationships were implemented as ordinary relational database primary key / foreign key entity relationships. To elaborate: having a class A and a class B in the object model meant having TableA and TableB in the relational model, with each row in the tables storing the corresponding objects' attributes. If there is a one-to-many relationship between class A and class B, we represent that relationship as a column in TableB which keeps primary keys from TableA. To access the B objects from an A object, we select all of the rows in TableB that reference the A object's primary key as a foreign key. Going the other way, to access the A object from any B object, we select the row in TableA that has the stored foreign key as its primary key.

For many-to-many bi-directional associations, we used join tables. A join table stores, separately from the referenced tables, the primary keys of the related entities. Extending the example above to a many-to-many relationship between A objects and B objects, this means an additional table, say TableAB, with columns A and B storing the list of keys of related A's and B's. To access the related B objects from an A object would mean executing a query that found

all of the related B objects' keys from TableAB, and then retrieved those rows in B that had those as primary key. Navigating the relationship from B's to A's is done similarly.

4.3.2   Inheritance mappings

According to [Ambler 2006], four basic approaches can be used to map object oriented inheritance structures to a relational model:

— Map the whole class hierarchy structure to one table, i.e. one table containing all of the combined attributes.
— Map each concrete class to its own table: ie all of the subclasses that inherit from an abstract class get their own table.
— Map each class to its own table.
— Map the classes into a generic table structure.

The Hibernate supporting literature provides strategies to enable one to use any of the approaches mentioned above [Hibernate 2006]. In the OO7 Java Hibernate code, in general, each class was mapped to its own table. This 'table per class' strategy [Hibernate 2006] was selected because it is a straight forward one-to-one mapping. The strategy of mapping *concrete* classes to their own tables was used only in one place, namely to map the OO7 abstract class called DesignObject and its subclasses: Module, Assembly, CompositePart and AtomicPart.

The mapping of concrete classes approach generally decreases the number of tables that are needed, compared to the table per class strategy. However, this approach results in duplicated information, since the properties of the parent class have to be included in each child class's table [Ambler 2006]. If properties are added later to the parent class, then they must be inserted into each child class's table.

One should keep this complexity in mind when choosing mapping strategies [Ambler 2006]. As the benchmark scenario did not include the addition of properties, this complexity was not an issue. Furthermore, the properties in child classes were quite simple: a build date, a string type and an id. (Note: it was necessary to rename the original 'id' field to 'design_id' as there were some conflicts in Hibernate with the name 'id'.)

4.3.3   Impedence mismatch difficulties

These mapping complexities are necessary to overcome the impedance mismatch between the object and relational models. In general, it is not trivial to map one-to-many relationships, many-to-many relationships, and inheritance structures, especially when the application is large and changing, and the mapping is manual. Special cases also complicate the mapping. For example, mapping a class that has more than one association to the same class is not handled by the simple scheme explained above. For example, in the benchmark model, BaseAssembly has two many-to-many mappings with CompositePart, called componentsPrivate and componentsShared It was decided to map these by using a separate join table for each relationship. Two join tables were therefore created, called *components_private* and *components_shared,* each containing the primary keys of the related BaseAssembly and CompositePart objects.

It is clear that this kind of manual mapping requires fairly intimate knowledge of both the object and relational models, and the mechanics of the chosen mappings. For example, when mapping AtomicParts and their Connections, a problem was encountered that related to order in which the objects had to be saved. This was manifested as a referential integrity constraint in the database. (Elmasri and Navathe [1994] describe a referential integrity constraint as the requirement that a tuple in one relation may not refer to a tuple in another relation unless the latter tuple *already exists*.) This meant that an AtomicPart object had to be saved first before saving a Connection object that is linked to it.

4.4   Overall observation

When creating an implementation of OO7 for a specific persistence mechanism, there is bound to be uncertainty about the correctness of the model that is saved. There is, of course, no magic bullet – no automatic way to check the correctness - one has to rely on testing. Thus, when the Module (the parent class in the OO7 benchmark) and its entire hierarchy tree are saved, the ideal would be that there should be certainty that the whole tree is correctly saved and retrieved. Unit tests for all of the operations that modify the model are required to enhance confidence that these individual parts are working correctly. Although this is envisaged at a later stage, for the interim we inserted counters to both implementations to count the number of objects created. We then opened each database and checked to see if the same number of objects existed.

5.   RESULTS AND OBSERVATIONS

The results of running *the small* configuration of the OO7 benchmark with *three connections* will be provided and discussed in the following sections. We also included in-memory implementation times to create and traverse objects, to give an indication of the times taken for non-persistence-related operations. However, the operations for queries and modifications were not implemented for the in-memory version as yet. This will be added in later versions if found to be relevant. The results for the implementations are grouped and reported below by the type of operation performed.

5.1   Creation

The time taken by each implementation to create and store the OO7 model is given in Figure 1. It was found that Hibernate took almost 2.5 times longer to create and store the model than did db4o. This is a significant difference.

Because object-relational mapping has quite a significant overhead, if you choose it as your persistence mechanism, you need to be sure that benefits you will enjoy because the data is now in relational form and therefor open to reporting and data mining tools, should be far greater than the overhead you will incur. Although a preliminary and plausible explanation is that this difference represents the object-relational creation and translation overhead (Hibernate needs time to create the tables, and map between the newly created objects and the tables), more detailed underlying reasons have not yet been established, and require further investigation.
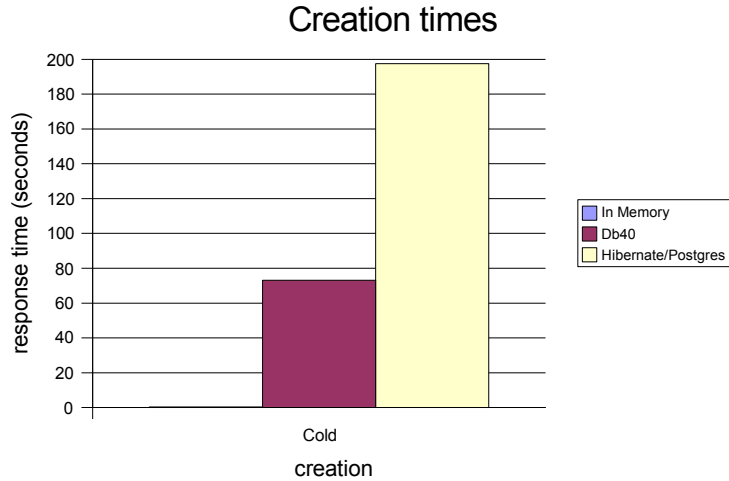
Creation times

Figure. 1. *OO7 Model creation times*

5.2   Traversals

Traversal times are a measure of the time that it takes to navigate around the entire object model. As previously mentioned, some of the traversals also modify the model by calling methods on the objects. Figure 2 displays the results for certain traversals, labeled T1, T2a, T2b, T2c, T3a, T3b, T3c, T6, T8 and T9. T4, T5 and T7 were not included as they were also not included in [Carey et al 1993a]. Carey et al [1993b] found that these did not provide any further insight into the performance of traversals and were left out of the original results. These missing traversals will be investigated in future work to see if they might provide interesting results on these new implementations.
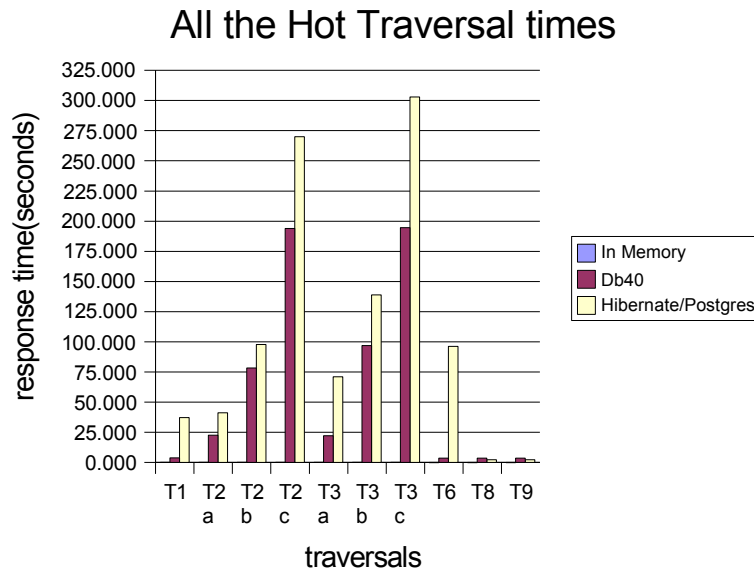
All the Hot Traversal times

Figure. 2. *Average hot traversal times*

It is clear from Figure 2 that, except for the cases of T8 and T9, db4o performs traversals through the object hierarchy more efficiently than Hibernate. T8 and T9 are traversal operations to find the Module and its associated Manual, respectively. These are very short operations, as there is only one Module in the small database with 3 connections. Thus these two traversals do not traverse the tree, but merely scan the Manual text string. This suggests that traversing the model is the significant difference between the two.

Singling out a couple of results for comment, Hibernate is approximately 100 seconds slower than db4o for traversals T3C and T6. Traversal T3C is based on T2C. It involves four traversals through all the atomic parts as well as

method calls to modify the date, either by incrementing or by decrementing its value. T3C in the original OO7 version tested indexing, which was included on the date objects. Recall that the Java version excluded indexing as a performance enhancement mechanism. T3C thus basically traversed 40 000 objects, calling methods on them. Again, it would seem, therefore, that Hibernate's slower overall time in this traversal test relates specifically to accessing the objects in a traversal operation.

Traversal T6 traverses the assemblies of the object model that are private and then traverses their root atomic parts. In total, 493 root atomic parts are traversed. This is considerably lower than the number of objects traversed in T3C, yet the time difference between the two persistence mechanisms is about the same. This suggests that Hibernate traversals are specifically slowed down when accessing lower levels of the object's model.

5.3   Queries

Queries were used to find objects in the model that matched certain criteria. The queries can be classified as follows:
- Queries on one table/class type using id's to match: Q1.
- Queries with ranges: Q2, Q3, Q5.
- Queries that find all the objects of one type of class: Q7.
- Queries that find objects of type A and then traverse their one-to-many associations: Q4, Q5.
- Queries that find objects of type A and do a "join" with another object B using id's to do the matching: Q8.

All of the queries involve iterating through the results returned and sending back a number of objects found. Q6 was not in the original OO7 results as it also did not provide any insight into the performance. For this reason Q6 was also not included in our results. Future work might include it.

As can be seen from Figure 3, Hibernate performs well for Q1 and performs the worst for Q8 (in fact, Hibernate's time for Q8 has been omitted from the graph because it is right off the scale: 5747.6 seconds). Q1 generates 10 random id's and then tries to find the atomic parts that match those id's. It thus appears that Hibernate performs well in finding random objects of the same type using an id for doing the matching. It is reasonable to assume that this is due to the efficiency of simple queries in the relational database more than compensating for the object-relational translation, as compared to the navigation required by the object database in performing the same query.
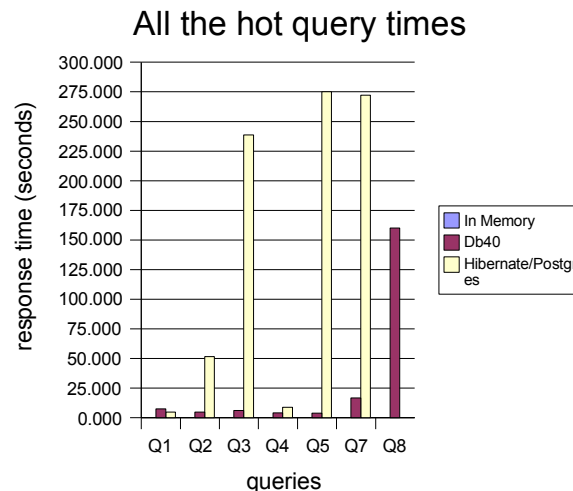

Figure. 3. *Average hot traversal times*

Q8 iterates through 10000 atomic parts and then finds the current atomic part's associated document. Q7 finds all 10000 atomic parts. Q3 is a query where the 10 000 atomic parts are found and only those that are in the range are returned. It would therefore seem from Q3, Q7 and Q8 that Hibernate performs badly when it needs to iterate through a large number of objects, retrieving and translating each one to its object form.

The slow performance of Hibernate for Q5 is strange as there aren't many objects involved. Q5 does however involve finding objects on lower levels and this could be the reason why it is slow.

The general picture conveyed by Figure 3 is that db4o queries are fast and that Hibernate is competitive only in isolated cases, where perhaps the performance of relational database part more than compensates for the object-relational overhead. Hibernate seems generally inefficient in performing queries that require joins.

5.4   Modifications

Section 2 mentioned that one of the types of OO7 operations is called a modification. The modifications in OO7 are known as structural modifications [Carey et all 1993] and consist primarily of inserting and deleting objects in the model. The results for our two implementations are displayed in Tables 1 and 2 below. The insertion and deletion times

for db4o are clearly better than those for Hibernate. Such differences are not as dramatic as those encountered in the traversal and queries tests.

	Db4o	Hibernate
Type of Runs:		
Cold Run	6.310 sec	11.291 sec
Avg of Hot Runs	12.020sec	23.4 sec

Table. 1. Insert times

	Db4o	Hibernate
Type of Runs:		
Cold Run	6.831 sec	7.686 sec
Avg of Hot Runs	13.355	18.711 sec

Table. 2. *Delete times*

6. CONCLUSIONS AND FUTURE WORK

By using a benchmark, it has been possible to compare and gain insight into the performance aspects of two object persistence mechanisms (object databases vs object-relational mapping to relational databases), as represented by two popular Open Source implementations (db4o and Hibernate).

It was found that db4o's overall performance was better than that of Hibernate. Many of the test results seem to confirm our rules of thumb (here, that the overhead of object-relational translation causes ORM-based implementations to be consistently slower than staying in object form with an object database). However, a few need deeper investigation (for example when the object-relational overhead is insignificant compared to the efficiency of relational query handling). When is this the case, and how can we design applications to take advantage of this?

During our investigation, it came to light that one needs in depth knowledge about these mechanisms to use them correctly and efficiently, with continued reliance on vendor input.

Creating the implementations of OO7 for each of these persistence mechanisms, it was found that it was possible to get the mappings and the storing of the object model wrong, and that this incorrect behaviour is difficult to identify with a cursory check of the code and data. An example of where this happened in our testing was with the mapping of a particular many-to-many relationship, where the combination of incorrect creation of objects and incorrect mappings caused corruption of the relational model. While this perhaps points more to the difficulties of implementing persistence correctly due to the complexity involved, in the case of a benchmark, the fact that implementations can be wrong means that wrong conclusions can be drawn. This indicates the absolute necessity for an auditing process to check the results of implementation creators and vendors.

Interestingly, object-relational mapping wasn't the only one at fault. The nuances of lazy loading and setting activation depth also caused some incorrect retrieval of objects until the mechanics were well understood.

Thus, each technology had its unique issues. Some common issues also surfaced, for instance it was not clear when to use cascade on update, when to use lazy loading, and when to store the objects in the database. An interesting issue was also found with Hibernate needing to re-associate an object that was returned from a query with a new session if the session was closed after the query run. Db4o did not run into this issue. Perhaps the answer is to be found in the depths of the age-old impedance mismatch, here extending to the differing ideas of sessions in the two worlds.

One of the original guidelines that had been decided upon as a basis of our investigation was to use the features of db4o and Hibernate in their basic, out of the box state, and not to use any optimizations. This was primarily to avoid optimization wars and specific tweaks that would not be obvious to the beginner user. However, some optimizations needed to be made, and parameter settings recommended, to make the comparisons meaningful. For example, using the current db4o version the authors did try the cascading functions and found that using too many cascades degraded performance, while using well selected cascades improved performance. We do intend to extend the scope of our research to more advanced features and optimizations, for example indexes and caches, in the future.

This paper reports on the first findings of our investigation into the performance of object persistence technologies. Future work will included comparing other technologies, like the persistence aspects of the new JDO 2.0 and EJB 3.0 specifications. We also intend to provide a test framework for OO7 to verify that it has been correctly implemented and that the correct operations are performed in all steps.

The random connection of AtomicParts, BaseAssembly and Composite parts needs some investigation, for the simple reason that the same object model should be used in each case for fair comparisons.

Further, the authors would like to investigate and compare the findings in using some of the other benchmarks, especially the new PolePosition [PolePosition] benchmark. Investigation into issues at more of an architectural level

should also be investigated, for instance a comparison between distributed (client-server) and single machine implementations, and a multi-user benchmark [Carey et al 1994].

We need to update our version of the OO7 Benchmark to create larger databases, as hardware has improved and the database sizes in the original benchmark are small compared to database sizes of today. Future work will also investigate the scalability of these implementations.

7.   REFERENCES

AMBLER, S.W. 1998. *Building Object Applications That Work Your Step-by-Step Handbook for Developing Robust Systems With Object Technology.* SIGS Books/Cambridge University Press, New York.

AMBLER, S.W. 2006. *Mapping Objects to Relational Databases: O/R Mapping In Detail.* www.ambysoft.com/mappingObjectsTut.html

ANDERSON, T., BERRE, A., MALLISON, M., PORTER, H., AND SCHNEIDER, B. 1990. The HyperModel benchmark, In *Proceedings Conference on Extending Database Technology*, Venice, Italy, March 1990, F. BANCILHON, C. THANOS AND D. TSICHRITZIS, Eds. Springer-Verlag Lecture Notes 416, 317-331.

CAREY, M. J., DEWITT, D.J, NAUGHTON, J. F. 1993a. The OO7 Benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, Washington, D.C, United States, May 1993, PETER BUNEMAN AND SUSHIL JAJODIA, Eds. ACM Press, New York, NY, USA, 12-21.

CAREY, M. J., DEWITT, D.J, NAUGHTON, J. F. 1993b. The OO7 Benchmark. CS Tech Report, University of Wisconsin-Madison, April 1993.

CAREY, M. J., DEWITT, D.J, NAUGHTON, J. F. 1994. A Status Report on the OO7 OODBMS Benchmarking Effort. In *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Language, and Applications*, Portland, Oregon, United States, 1994, ACM Press, New York, NY, USA, 414-426.

CATTELL, R.G.G, BARRY, D., BERLER, M., EASTMAN, J., JORDAN, D., RUSSELL, C., SCHADOW, O., STANIENDA, T., VELEZ, F., 2000. *The Object Data Standard: ODMG 3.0.* Morgan Kaufmann Publishers, San Francisco.

CATTELL, R.G.G, 1991. *Object Data Management: object-oriented and extended relational database systems*, Addison-Wesley Publishing Company.

CATTELL, R.G.G, SKEEN, J., 1992. Object Operations Benchmark, *ACM Transactions on Database Systems*, Vol. 17, No. 1, 1-31.

DB4O. 2006. *Db4o Tutorial*, http://www.db4o.com

ELMASRI R., AND NAVATHE S.B. 1994. *Fundamentals of Database Systems.* The Benjamin/Cummings Publishing Company Inc, Second Edition.

HIBERNATE. 2006. *Hibernate Reference Manual.* http://hibernate.org

JORDAN, M 2004. *A Comparative Study of Persistence Mechanisms for the Java™ Platform, September 2004. http://research.sun.com/techrep/2004.*

OZONE, http://www.ozone-db.org

POLEPOSITION, http://www.polepos.org