The Usage and Performance of Object Databases compared with ORM tools in a Java environment

Mikael Kopteff
Information Technology
Haaga-Helia University of Applied Sciences
and
Floobs Ltd.*

ABSTRACT

Object databases have been almost entirely forgotten in the mainstream software development world. The huge interest in object databases during the 1990's has changed to ignorance even though object-oriented programming languages like Java and C# continue to thrive. During the last couple of years many object-relational mapping tools have emerged to solve the notorious object-relational impedance mismatch between relational databases and object-oriented languages. The widespread popularity of object-relational mapping (ORM) tools still raises the question of using an object database instead of a redundant data mapping tool and persisting data in its natural form – as objects.

The goal of this work is to find out if object databases can be used as persistent storage in applications instead of ORM tools and relational databases. Two popular tools from both camps were selected as reference implementations for the study. The features, usage and performance of these tools were then studied from a software development point of view. Features of both tools were listed and compared. Usage of the tools was studied by comparing the query languages and the actual code used to access the database. Performances of the two reference implementations were tested by running a subset of the de-facto object database benchmark OO7. These three factors provide sufficient information if object databases can replace ORM tools and relational databases as the persistence layer in modern software development.

Keywords: object-oriented databases, database (persistent) programming languages, relational databases, performance evaluation, object-oriented programming

Author address:
Mikael Kopteff (miku.kopteff(at)gmail.com), Haaga-Helia University of Applied Sciences, Finland (www.haaga-helia.fi)
*Floobs Ltd. (www.floobs.com) provided funding for this research.

1. INTRODUCTION

Modern day software development, especially in the business sector relies heavily on relational databases, mostly because of their reliability and standardised query language. Using object-oriented languages with relational databases to develop applications raises the impedance mismatch (Cattell 1991: 122) between the two models in use. In this type of situation, the designer is forced to compromise regarding the implementation of the application. The object-relational impedance mismatch between the object-oriented programming language and the relational database can be solved by many different ways, for instance converting objects to the relational model, using an object database or partly converting objects to the relational model (Dietrich & Urban 2005: 104). This study investigates if object databases can be used as replacements for automated object-relational mapping tools and relational databases in a Java environment and concentrates heavily on the implementation details of the two tools used in the comparison. The approach taken is more from the point of view of the programming language, rather then of databases.

2. REFERENCE IMPLEMENTATIONS

The two implementations selected for closer study were chosen, since they are mature products, they both support a standardised interface for Java applications and both tools have been benchmarked with OO7 before.

2.1 Versant ODBMS

Versant is an object database, that has over 50 000 users in different fields, like telecommunications, finance, defence, government, simulations and medical (Versant 2007b). Versant was also part of the original OO7 benchmark (Carey et al. 1994).

2.2 Hibernate ORM

Hibernate is a popular open ORM tool, which supports JPA (Java Persistence API) and the JPA specification is partly based on Hibernate (Bauer & King 2007: 31). Hibernate is always used with a relational database and in this study it is used with Oracle 10g Enterprise Edition.

3. COMPARISON

This section covers the comparison performed between the two tools. The features selected for this comparison are persistence-related features often found in Java applications. The comparison of usage and performance provides a more in-depth look into both implementations.

3.1 Features

A comparison of full features would be pointless between these two tools, since Versant ODBMS is a full database and Hibernate ORM only provides tools for object-relation mapping and not the actual database functionalities. These tools can be still compared from a software development point of view and the investigated features are the services and interfaces provided for applications written in Java programming language. Table 1 lists the features compared.

Table 1: Comparison of software development related features between Versant and Hibernate (Hibernate 2007, Versant 2006a, Versant 2006b, Versant 2006c ja Versant 2007a).

FUNCTIONALITY	VERSANT ODBMS 7.0.1.3	HIBERNATE 3.2.5
Java interfaces	JDO 2.0, Java Versant Interface (JVI Transparent and JVI Fundamental)	JPA 1.0, Hibernate API
Bi-directional relationships	Yes	Yes
POJO persistence	Yes	Yes
Transitive persistence between objects	Yes (JVI Transparent, JDO)	Yes
Support for classes in Java Collections -framework	Yes, also for ODMG (Object Data Management Group) 2.0 collections	Yes, supports other collections, like Bag
Locking mechanisms	Optimistic and pessimistic locking	Optimistic and pessimistic locking
Database access control	JTA, JNDI	JTA, JDBC and JNDI
Connection to EIS	JCA	JCA (experimental)
JTA support	Yes	Yes
Automatic session control	Yes	Yes
Optimization	Caching, fetching strategies	Caching, fetching strategies
Distribution	Yes	Yes, (with Hibernate shards)
Dirty checking	Yes	Yes
Automatic generation of schema	Yes	Yes
Query mechanisms	JDOQL, SQL, VQL	HQL (JPA QL), Criteria API, database specific SQL
XML support	VXML for converting objects to XML and back	Yes (experimental)

Both tools provide more than one interface to access the database from the Java programming language. Versant supports the newest version of the domain model persistence specification JDO API, while Hibernate supports the object-relational mapping specification JPA. Both of these specifications are products of the Java community process. Versant also provides a Java Versant Interface (JVI) API, which also supports the ODMG 2.0 standard. JVI has two sections, the Fundamental API, which is a straight wrapper to the Versant C API and a higher level Transparent API, which provides transparent persistence, ODMG support and automatic binding of database objects to Java objects. Hibernate also provides a non-standardized, Hibernate specific API. In both Versant and Hibernate the non-standardized API provides more functionalities than the standardised API.

In general both tools have very similar features and they provide similar services for applications to use. Both tools provide services like JCA (Java EE Connector Architecture), JTA (Java Transaction API) and automatic session control, which all can be considered vital in modern day Java development. Both tools even provide advanced features like “dirty checking” (Bauer & King 2007: 49), where the already stored objects are automatically updated into the database after modification.

3.2 Usage

One of the main differences between developing software with an ORM tool and developing software with an object database, is that ORM tool converts the data into relational form, while object databases store the data usually as it exists in the object in the programming language (of course there is mapping to a certain degree in object databases too). While working with Hibernate, all design and implementation decisions should be done with this fact kept firmly in mind, since a large portion of work persisting objects is done by the underlying database engine. The designer should have a very deep understanding about object-orientation, the relational model and the database engine used. When running an application using an ORM tool, part of the errors come from the underlying database engine and this is why Hibernate is not a fully independent middleware tool. Different database

engines can control integrity differently or they can store data types differently and hence the functionalities of Hibernate become database dependent.

Another big difference between Hibernate and Versant is object identity and the way it is dealt with. In the relational world object identity is dealt with primary keys and in the object world by object identifiers (OID) (Cattell & Barry 2000: 17). Bauer and King (2007: 16) suggest using surrogate keys with Hibernate, which creates an artificial object identifier (similar to an OID), which is not used in the applications, but still exists as a property of a class.

3.2.1 Schema

When creating a persistable class for Hibernate, the creator has to consider the inheritance strategy, bi-directional relationships, the object/table identity and the way the properties of a class will be mapped to the tables and columns. A good example arises, when storing a Java `String` property of a class to a relational database `varchar` data type, which has a length of 256 characters. While the usage of the property works in the application if the length of the property is longer than 256, the database engine will throw an exception when trying to commit the data to the database, since the data length is too long for it. When creating a persistable class in Versant the creator only has to consider the actual object model. Of course, extra features like cascade rules or the bi-directional relationships can be also defined. Versant automatically converts Java data types to Versant supported data types according to either JDO or ODMG specifications depending on the interface in use. Versant can create the database schema from the JDO XML metadata file, programmatically from the application or using Java classes depending on the interface in use. When using JVI interface, the class definition is created in the schema, when the class is persisted for the first time. In Hibernate the database schema is created either from the XML metadata files, Java annotations or by using DDL.

The following simple example demonstrates the usage of Hibernate XML mapping files for an Employee class that has a many-to-many relationship with a Project class, an indexed “name” property and some other properties.

```
<hibernate-mapping>
    <class name="example.model.Employee" table="EMPLOYEE">
        <id name="id" column="EMPLOYEE_ID">
            <generator class="native"/>
        </id>
        <property name="name" index="name_idx"/>
        <property name="salary"/>
        <property name="title"/>

        <set name="projects" inverse="true" table="PROJECT">
            <key column="EMPLOYEE_ID"/>
            <many-to-many column="PROJECT_ID"
                  class=" example.model.Project"/>
        </set>
    </class>
</hibernate-mapping>
```

The example that follows is the exact same class defined with Versant's JDO XML metadata file.

```
<jdo>
    <package name=" example.model">
        <class name="Employee">
            <field name="name" indexed="true"/>
            <field name="salary">
            <field name="title">

            <field name="projects">
                <collection element-type="Project">
                    <extension vendor-name="versant" key="inverse"
                                value="employees"/>
                </collection>
            </field>
        </class>
    </package>
</jdo>
```

When comparing these two simple examples, they seem to be very close to each other. The notation is similar and the JDO version merely lacks the relational database mapping options and the rest of the differences are mainly semantic. For example, the `property` keyword is replaced with the `field` keyword and the mapping of an inverse relationship in Hibernate requires the foreign key column name. When defining a schema with Hibernate, the identity of an object should be declared. In this example the database engine handles it by declaring the `generator class="native"` definition for the `id` column. Of course, in Hibernate one could also add maximum lengths of columns and other types of object-relational mapping information to the XML mapping file.

Hibernate can also define the database schema using Java annotations and this type of definition can be compared with Versants way of defining the schema using the structure of a Java class. In both ways, the class structure itself is the definition for the class in the database. In this situation, Hibernate uses annotations to provide the information for the object-relational mapping.

When altering the schema, Versant uses a schema generation strategy and a tool (Versant 2006a). In Hibernate, the schema can be updated using hbm2ddl tool (Bauer & King 2007: 40), which can be also used in the schema generation. It is important to notice, that both Versant and Hibernate provide tools for automatically generating or updating the schema and the use of DDLs is beginning to seem out of date, even though still possible.

Even though the basic definitions are very similar in both tools, Hibernates definitions provide more options, partly because of the relational database engine, that needs additional information. In Hibernate API, users can more accurately define fetching strategies and cascade rules. One big difference in the logic of the tools is, that when using Versant JDO, the properties of classes, that are not defined in the metadata file are also persisted by default, while in Hibernate only properties found in the XML mapping file are persisted. This is inconsistent however when using Hibernate annotations, since properties, which do not have any definition, are persisted by default.

3.2.2 Connections and Transactions

Hibernate and Versant both use session as a unit of work and all events, which handle persistent objects are performed through sessions. In both tools a single session can also include several transactions. The following section takes a deeper look into the code of transaction management.

The following example shows how to create a session using the Hibernate API. As seen from the code, the session is created first and the transaction is requested from the session. A new session is acquired from the `SessionFactory` class.

Hibernate API

```
SessionFactory sessionFactory;
try {
    sessionFactory = new Configuration().configure().buildSessionFactory();
    session = sessionFactory.getCurrentSession();
    session.beginTransaction();
    session.getTransaction().commit();
    sessionFactory.close();
} catch (Throwable ex) {
    //Error handling
}
```

As mentioned earlier, JVI has two different types to access the database. The following example shows how to access the database using the Transparent API. Transactions are accessed through sessions and no directions are given straight to the transaction object.

Versant Transparent API

```
Properties prop = new Properties();
try {
     in = VersantManager.class.getResourceAsStream("/versant.properties");
     prop.load(in);
     TransSession session = new TransSession(prop);
     session.commit();
     session.endSession();
} catch (IOException e) {
    //Error handling
}
```

When using the standard JDO and JPA interfaces, the transaction management is handled almost similarly in both tools. Using the ODMG interface in Versant is much more straightforward, but the basic concepts do not differ much. The following examples show the basic opening of a session and a transaction in Hibernate JPA and Versant JDO. As seen, the differences between tools are mainly semantic.

Hibernate JPA

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory(“example”);
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
em.getTransaction().commit();
em.close();
```

Versant JDO

```
PersistanceManagerFactory pmf =
     JDOHelper.getPersistenceManagerFactory(“example”);
PersistanceManager pm = pmf.getPersistenceManager();
pm.currentTransaction().begin();
pm.currentTransaction().commit();
pm.close();
```

Saving objects is very similar in both tools since it is done by calling a single method (`save(Object)` in Hibernate API and `makePersistent(Object)` in Versant Transparent API). Transitive

persistence can be achieved with the same operation in both Versant and Hibernate. One major difference is that Hibernate can set an association to be bidirectional automatically (since relationships are bidirectional in the relational model), while in Versant the other endpoint of an association must be set manually to make traversing the relationship from both directions possible.

The basic concepts in both tools are similar regardless of the interface in use. Both tools use some type of setting file, which is loaded and used as a base for the database connection. The fact that both tools use sessions as units of work, makes the using of the APIs very similar.

3.2.3 Queries

Both tools have several query languages partly depending on the interface in use. Hibernate has HQL, that is similar to SQL (Bauer & King 2007: 533), but supports object features. Hibernate also has the Criteria API, that provide advanced query capabilities. Versant has a Query API that includes VQL and predicate queries for the JVI interface. VQL is a subset of OQL (Versant 2006a). For the JDO interface the JDOQL (JDO Query Language) is provided for querying.

In HQL, referring to child objects is done by using dot notation, same as in OQL. Hibernate also includes the Criteria API that lets you create queries using Java classes and methods. This is partly similar to JDOQL or the predicate queries used in Versant.

The following examples compare the creation of a simple query with a single restriction. The query is created using the session and the query text is given as a parameter to the executing method. Parameters can be used in the query string and than bound to the query.

HQL

```
String nimi = "Foo";
Query query = session.createQuery("from thesis.model.Employee where name =
:name");
query.setParameter("name", name);
List<Object> objects = query.list();
for (Object object : objects) {
    //handle object
}
```

VQL

```
String name = "Foo";
Query query = new Query(session, "select selfoid from thesis.model.Employee where
name = $name");
query.bind("name", name));
QueryResult result = query.execute();
Object obj;
while ((obj = result.next()) != null) {
    // handle object
}
query.close();
```

Again both tools perform the tasks very similarly and the classes representing a query have the same name. In Hibernate the session creates the query, but in Versant it is passed as a parameter to the constructor of the `Query` object. The query result in Hibernate is a `List` object, but in Versant is a special `QueryResult` object. Both tools provide similar ways of binding parameters to the query. The actual query languages have some differences, like the `selfoid` keyword in VQL, which in is used when accessing the whole object while in HQL no select clause is needed. HQL uses dot notation when referring child objects, while VQL uses the arrow notation.

Criteria API and JDOQL have more differences, since Criteria API uses actual Java classes and methods to set restrictions for the query and JDOQL uses Java syntax inside a string variable. Even though the basic idea is the same, the implementations differ quite a lot. One of the most important features in Hibernate and Versant is the possibility to use SQL in the queries. This extends the possibilities of both tools, since SQL can be used to improve efficiency, for instance.

Table 2: Query methods in Versant and Hibernate.

FEATURES:	HQL	VQL	Criteria API	JDOQL
Syntax	SQL and OQL based	SQL and OQL based	Pure Java	Java syntax in a string variable
Navigation to attributes	Dot notation	Arrow notation	Java	Using variables
Use of variables	Yes	Yes	Yes	Yes
Navigation	Yes	Yes	Yes	Yes
Aggregtion (sum, average, etc.)	Yes	No	Yes	Yes

Table 2 lists the main functionalities of the query languages and their usage. HQL and VQL are very similar in syntax and in functionality. Criteria API and JDQL have similar uses and the basic principles are the same, but they still have lot of differences. Hibernate and Versant can both fulfil the needs for querying especially since both tools have support for traditional SQL.

3.3 The OO7 Benchmark

One of the most significant benchmarks concentrating on object databases is the OO7, which was first published in 1993. The OO7 benchmark doesn't produce a single interpretable number, but several sets of results. The actual tests can be divided into three categories, which are traversals, queries and structural modifications (consisting of inserts and deletes) (Carey et al. 1994). The original benchmark was implemented using C++ and it was performed on four different object databases. The OO7 has also been used to compare object databases and ORM tools by Van Zyl et al. (2006). In the study Db4Objects, a lightweight database was compared with PostgreSQL and Hibernate.

The OO7 benchmark is very complex and the data model itself has a lot of inheritance and complex objects, because of its roots in CAD/CAM/CASE applications (Carey et al. 1994). The model can be loaded with different parameters resulting in three databases with different sizes and different amount of objects. The root of the entire class model is the abstract `DesignObj` class, from which most of the classes inherit basic properties. The class model can be roughly divided into two sections, which have a certain role in the model. The first one is the design library and the other the assembly hierarchy. Assembly hierarchy consists of seven levels of `Assembly` objects that are connected with other `Assembly` objects (Carey et al. 1994). The assembly hierarchy is less interesting from a performance perspective and far less used than the design library. The design library is connected to the assembly hierarchy through `CompositePart` objects. `AtomicPart` are the objects most used in queries and they are connected to `CompositePart` objects.

3.3.1 Modifications to the OO7 benchmark

The OO7 is a massive benchmark, which provides enormous amounts of data to be interpreted. Because of this the benchmark was not implemented fully and the tests where ran only against a medium sized database. Also, changes where needed since the original OO7 was implemented in C++ and the approach here was not similar taken by Van Zyl et al. (2006), where the purpose was to port the existing C++ code to Java. In this study, the implementation was made from scratch and code from other implementations was not used. The implementation was based only on the descriptions of test cases given by Carey et al. (1994) and for this reason the results are not fully comparable with the results from the other OO7 implementations.

The Versant implementation was made by using JDO and the Hibernate implementation using both JPA and Hibernate API annotations. The inheritance mapping strategy used was table per class, since the `DesignObj` was an abstract class and never implemented, so there was no need to map it to a table. The queries in Hibernate were mainly implemented using Criteria API, but some using HQL, since in some cases HQL was more convenient and more flexible to use. Versant queries were implemented using JDOQL. In this study the tests were only ran against the medium sized database.

A large portion of the traversals (navigating the object graph) was omitted from the benchmark, since detailed traversal and update efficiency was not considered important. Traversal 1 and 2 were also made lighter and they only retrieved the first 20 objects from the `BaseAssembly` objects. None of the fields were indexed, except the `buildDate` attribute from the `Document` class. Because of this, traversal 3 was also omitted from the tests, since it does not provide any additional information unless performed on objects with indexes. This was replaced with a traversal X, which performs a simpler update. The most interesting point on indexes is that Oracle 10g creates indexes on all primary key columns. This fact raises an interesting question concerning indexes, when dealing with the object relational impedance mismatch. In this study, the `id` field was used to maintain unique objects, resulting that the object identity was indexed automatically. Should in this situation all Versant id columns also be indexed? Versant maintains object identity using LOID (an own implementation of OID) and these cannot be indexed, since they are not properties and so indexing a field in a Versant object is not the same if a field is indexed in Hibernate (or Oracle). Also, the automatic generation of indexes on primary keys is an internal function of Oracle, so the creation of indexes in Oracle is not user created optimization, but the creation of indexes in Versant is. Due to all these reason, indexes were almost unused in the benchmark, except in traversal X. Major reason was also, that indexes are used in optimization and in this study the idea was to use default settings as much as possible, even though OO7 specifies indexes.

Query 4 was also omitted because of the indexes. Each of the test cases were ran in separate transactions. Traversals 8 and 9 were omitted, because of a problem with the Oracle JDBC thin driver when accessing CLOB field over 4000 characters long. Traversals 8 and 9 were also not vital to the benchmark since large text fields are rarely stored in a database.

3.3.2 Testing environment

The benchmark was executed with two Hewlett-Packard PCs that were identical. Single machines hardware consisted of a Core 2 Duo processor, 4 GB of DDR2 667 MHz RAM, 500GB hard disk (7300 rpm). Both machines ran MS Windows XP Pro, SP 2, which were installed from the same image. The Java runtime used was Sun's Java VM 6, update 2. All tests were performed in a computer laboratory, with no other network traffic. Machine A had Versant ODBMS installed and machine B Oracle 10 g. The server machine was always called from a client machine and for Versant, machine B was the client and for Oracle, machine A was the client. During the tests, Java VM was provided with 1024 MB of memory at runtime. Both databases were installed with default settings and parameters.

3.3.3 Benchmark results

While interpreting the results it is important to keep in mind, that benchmarking is always dependent on many variables and that all result only reflect the results of the tests in the benchmark. Even though the tests in the benchmark endured a lot of testing, they are error prone, just as any other code. To avoid errors, tests in both tools where based on the same code base. Also it is important to remember, that the OO7 is developed for object-oriented databases using the object model and as a result the domain model of the benchmark has lot of features like inheritance and many-to-many relationships, which are more complex to handle in relational databases (Bauer & King 2007: 17). It is also good to keep in mind, that in Hibernate for each test case, a new `SessionFactory` object is created, which is

an expensive operation. The test times displayed here are “cold” executions (cache empty) and from the first run of each test case.

Traversal 1_MOD measures raw traversal speed and the test browses the 20 first `BaseAssembly` objects, their `CompositePart` objects, their `AtomicPart` and for these objects a DFS (Depth First Search) is performed. Traversal 2 A_MOD performs the exact same functions than Traversal 1, except it also updates some objects. Traversal 6 is the same than Traversal 2, but does not perform the DFS. Traversal Cached Update performs Traversals 1 and 2 in the same transaction, so the objects should be cached in the session. In Traversal X all `BaseAssembly` objects, their `CompositePart` objects and their `Document` objects are traversed. The text fields of `Document` objects (that are indexed) are then updated. The performances of the traversals are shown in charts 1 and 2.

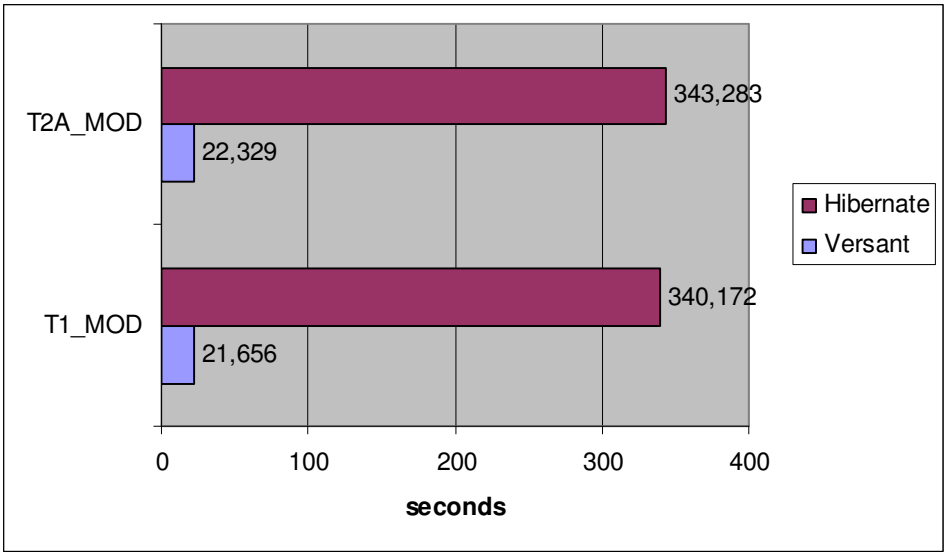

Chart 1: Traversals 1_MOD and 2 A_MOD

Traversals 1_MOD and 2 A_MOD perform similar tasks and the difference between Hibernate and Versant can be the result of memory management or the mapping strategy used in the tables. Interesting is, that Traversal 2 A_MOD did not take much more time to execute, than Traversal 1_MOD even though it updates objects.

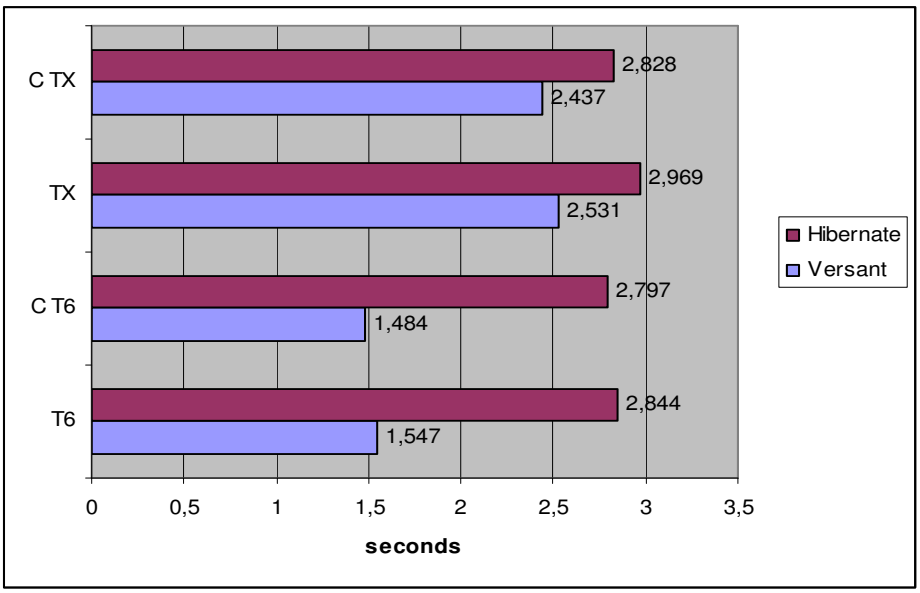

Chart 2: More traversals

Rest of the traversals don't have so much difference in the execution times, traversal 6 is twice as fast in Versant than in Hibernate, but Traversal X, which updates indexed columns, is almost equally fast.

Query 1 finds 10 random `AtomicPart` objects and queries 2, 3 and 7 find a certain percentage from the `AtomicPart` objects. Query 5 and 8 compare properties of objects between two different classes. The performances of the queries are shown in chart 3.

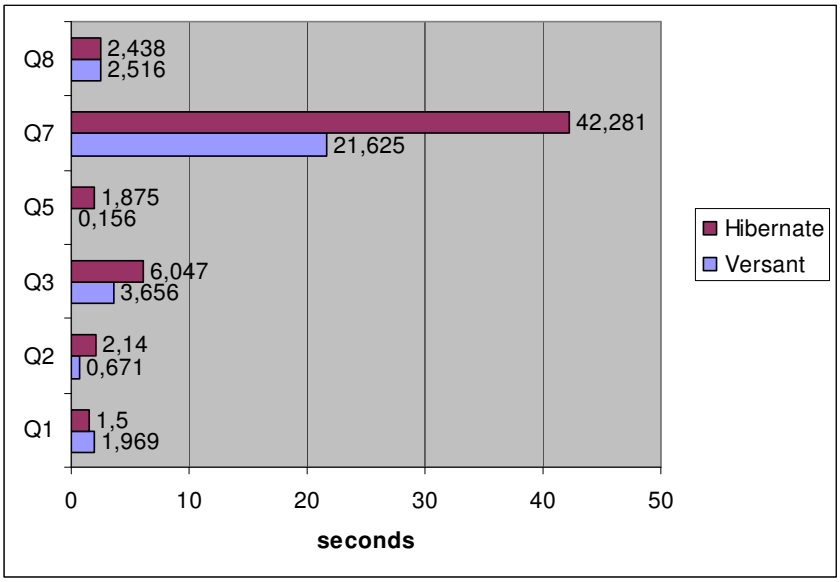

Chart 3: Queries

The results for queries 1 and 8 are easily explained, since both queries target an indexed column in Oracle (indexes, which Oracle automatically creates). It would be interesting to see how Versant would perform if these columns were also indexed in Versant, but of course this would slow down the inserts and updates. In rest of the queries Versant was clearly faster.

The final test cases were structural modifications, which removed approximately 1000 objects from the database. As seen from chart 4, the performance of the insert operation had a small difference and the delete operation in Hibernate took surprisingly long. When comparing this result to Traversals 1 and 2, it seems that Hibernate has problems handling large masses of data.

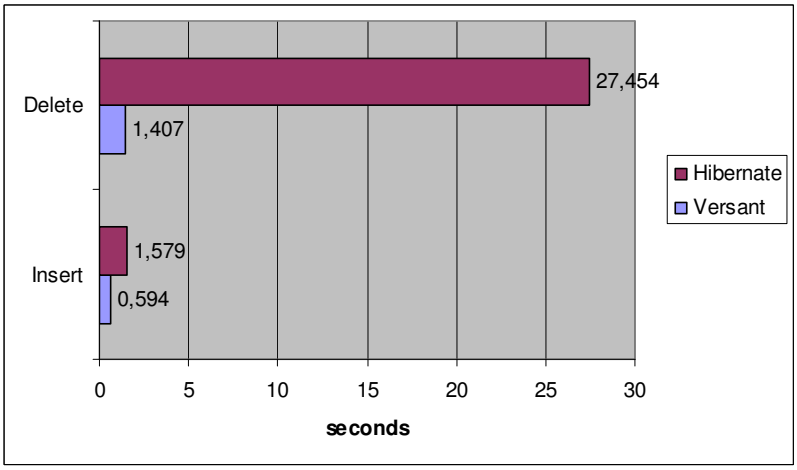

Chart 4: Structural modifications

Overall the results seemed to be consistent with previous research (Van Zyl et al. 2006), that object databases are faster in managing object style data. Most surprising was the long performance times in Hibernate while deleting objects, but this can be partly explained with the use of indexes, since Oracle has to maintain the indexes when performing these operations.

4. CONCLUSIONS

From the three areas investigated, features, usage and performance, Versant was surprisingly strong in all areas. When we consider the features in both Versant and Hibernate, they provide similar, and in some cases identical, services for users. From the usage section we can conclude, that using Versants interfaces is at least equally easy than using Hibernates interfaces and in most cases easier, since it lacks the object-relational mismatch configuration. Using an object database lets the developer focus on actual design problems and not on the problems of the ORM tool or the conversion of objects to relational form. When considering the results of the performance comparison, Versant was faster in most cases, hence it is more efficient to use. When looking at the results as a whole we can conclude that Versant is more useful for developing applications, since users can use Versant to develop applications more efficiently and with less workload, than with Hibernate. This also results savings in costs, which could be one of the key factors for the usage of object database to spread.

5. DISCUSSION

One major issue in the study was the benchmark used. The OO7 is somewhat outdated and the object database community is in need of a new database benchmark, which not only focuses on object features, but measures situations, that are common also in modern software. OO7 has still the historical baggage of CAM/CAD applications and the test cases have a great deal of traversals, and almost no complex queries. The OO7 benchmark can be easily criticized because of these reasons, and that also can crumble the trust for object databases.

6. ACKNOWLEDGMENTS

I would like to thank Timo Raitalaakso from Solita and Timo Westkämper from Mysema for their expertise. I would like also to thank Elina Huhtala from Solita, Jukka Juslin from Haaga-Helia University of Applied Sciences, Anssi Piiranen and Hannu Leinonen from Floobs for their comments. I would also like to thank Maria Ball from Versant Corporation for her assistance.

7. REFERENCES

Bauer, C. & King, G. 2007. Java Persistence with Hibernate,
Greenwich, CT: Manning Publications Co.

Carey, M.J. & DeWitt, D.J. & Naughton, J.F. 1994. The OO7 Benchmark,
In proceeding of the 1993ACM SIGMOD international conference on management of Data, New York, USA, 12-21.

Cattell, R.G.G. 1991 Object Data Management: Object-oriented and Extended Relational Database Systems
Addison-Wesley

Cattell, R.G.G & Barry, D.K. [et al.] 2000. The Object Data Standard: ODMG 3.0
San Diego: Morgan Kaufmann Publisher.

Dietrich, S. W. & Urban, S. D. 2005. An Advanced Course in Database Systems: Beyond Relational Databases, New Jersey: Prentice Hall.

Hibernate 2007. Hibernate 3 reference manual, accessed: 24.10.2007.
http://www.hibernate.org/hib_docs/reference/en/html/

Van Zyl, P. & Derrick, G.K. & Boake, A. 2006. Comparing the Performance of Object Databases and ORM tools, ACM International Conference Proceeding, Vol. 204, 1-11.

Versant 2006a. Versant Database Fundamentals Manual (Release 7.0.1.3)

Versant 2006b. Versant JVI API documentation (Release 7.0.1.3)

Versant 2006c. Java Versant Usage Manual (Release 7.0.1.3)

Versant 2007a. Versant JDO Interface User's Guide.

Versant 2007b. Versant. accessed: 5.11.2007.
http://www.versant.com/en_US/products/objectdatabase