Integration and evaluation of a low-level storage provider for persistent object-oriented data management

Integration and Evaluation of a Low-Level Storage Provider for Persistent Object-Oriented Data Management

Master Thesis

David Simmen

<dsimmen@student.ethz.ch>

Prof. Dr. Moira C. Norrie

Christoph Zimmerli

Global Information Systems Group

Institute of Information Systems

Department of Computer Science

14th September 2010

ETH

Eidgenössische Technische Hochschule Zürich

Swiss Federal Institute of Technology Zurich

globis

Abstract

Storage of object oriented data, still is one of the major challenges of todays developers. Although quite a number of solutions are available on the market, we believe that none of them really satisfies the needs of todays developers and researchers. In this thesis we will present a new solution using an almost transparent approach based on a new storage provider. The two main focuses lie on transparency capability and performance of the provider.

Contents

1 Introduction

Persistent data management on an object oriented programming platform such as Java requires the developer to decide, which of the available technologies to use. Today relational databases pared with an object relational mapping framework such as JPA [1] or Hibernate [2] are still the main technology used. This decision is based on the fact that most data is not only used by one application, but is used by a number of different applications and tools, and therefore developers accept the complications resulting from the generally accepted impedance mismatch.

However, object oriented databases or persistent programming languages are developed since the 1980s, trying to resolve this impedance mismatch. Unfortunately did the technologies which augmented existing languages such as as PJama for Java [3] or PS-Algol for Algol [4] not have the impact on the persistent technologies that they went for.

There exist a number of object oriented database management systems today such as db4o [20], Versant [18], ObjectStore [21] or Objectivity [22] which have received increased attention. Compared to the earlier attempt in which existing languages were augmented, these systems can be seen as additional elements. The technologies are available over a number of programming platforms such as Java, C# or C++ and support different levels of orthogonality and transparency. Since they typically adopt the object model of the programming language, they are not suitable to experiment with other models or approaches of integration.

Resolving the lack of object oriented storage technology which supports experimentation on a number of different levels such as transparency, a storage layer has been designed as part of the Avon project [5]. It consist of an API and different providers implementing it. Based on a simple model on top of which more complex models can be developed, the API declares a set of low-level storage operations.

The main goals of this thesis are to evaluate the usability of the API in context of transparency and performance. We will compare different kinds of integration methods, benchmarks and storage providers, to evaluate usability and performance of the storage layer.

In chapter 2 we will first have a look at object oriented databases in general and then will have a look at the proposed storage model and the storage API in chapter 3. Chapter 4 covers the issues and solutions concerning the integration of the storage provider including transparency capability and enhancement and inspects the performance of the proposed solutions. In chapter 5 we will have a closer look at provider implementations and propose different new approaches. This thesis is conclude by a brief discussion of the results and propositions for future research direction in chapters 6 and 7.The implementations in this thesis were done entirely in Java and therefore are all the code listings included in this thesis are written in Java.

2 Object-Oriented Databases

Object oriented database management systems (OODBMS) have to cope with a number of different requirements. We will have a look at orthogonality and transparency which are considered apart from performance, two of the main requirements.

2.1 Orthogonality

Orthogonality is often discussed in the context of object persistence. Orthogonality as proposed by [6] means that every type of object can be made persistent. If for example a object database does not allow the persistence of public fields it would not be fully orthogonal. While this obstacle can be handled by developers by either using only private fields or adding code to cope with this limitation it still reduces the capabilities of the Java object model. We will have a look if our storage model and especially its API provide the mechanisms to store any possible Java object in chapter 4. As stated in [7] there are some major problems with orthogonality and Java, which are:

- Class objects
- static variables
- Threads
- Throwables

We will only consider static variables of the above with our storage API.

2.2 Transparency

Transparency in the context of object oriented data storage means that all of the persistence management code is hidden from the developer.

There are different approaches available from different vendors today. In Java, Versant enhances the byte code of marked classes after compilation, and Hibernate uses Annotations to map fields of a class to fields in a table of a relational database. Whereas the Versant approach is quite transparent, except for marking the persistent classes which is done in a separate configuration file, it provides some small difficulties when it comes to packaging and distribution of Java code. The Hibernate approach on the other hand is not really transparent since you need to map your fields manually and sometimes even have to create the tables in your database, it only frees the developer from writing SQL statements in its Java code.

2.3 OM and Avon

The OM model proposed by [8] is an extension for the ER model, to model typing and classification for object oriented data. The Avon project [5] is a Java implementation for the OM model. Figure 2.1 shows an outline of the different components of the Avon implementation, where it can be seen that the storage of objects is completely decoupled from the model components.

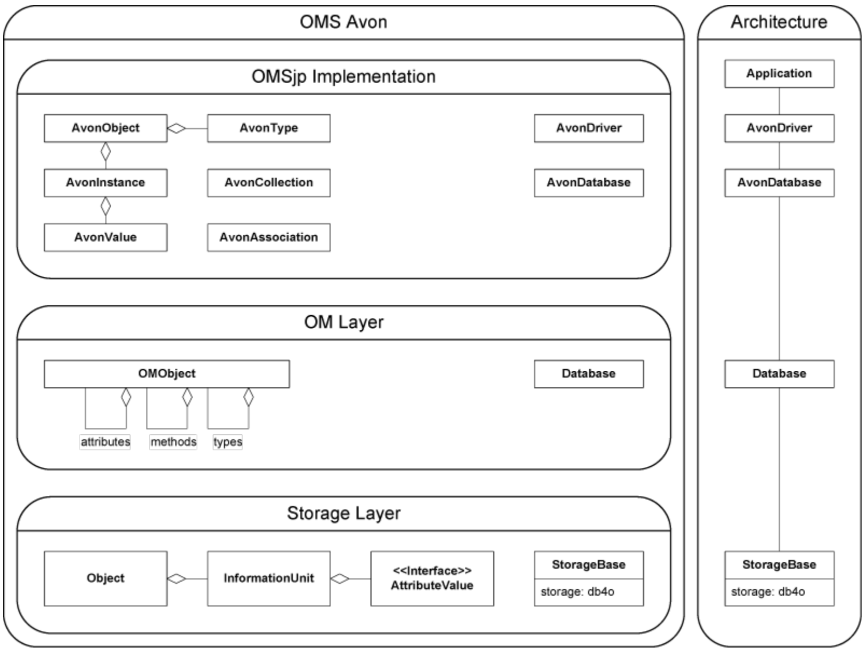

Figure 2.1: Architecture of the Avon implementation from [5]

2.4 Performance and Benchmarking

Since the storage provider not only has to be easier to use by developers and deliver a higher amount of transparency, but more important must also compare on a performance level with other products, we will benchmark it with other implementations, which mainly is db4o. Unfortunately there is nothing like the TPC consortium for OODBMS benchmarks. Therefore we had to search and compare a number of different benchmarks used in the OOSDBMS community.

2.4.1 Pole position and OO7

The pole position benchmark [9] is the one benchmark used by db4o and currently the one which gets the most updates in the object oriented database world. However, almost all of the changes are made by the db4o team and other implementations, like for example for Hibernate are probably not as fast as they could be since they are not implemented by their development team. But since the main competitor for the storage provider is db4o, this is a minor disadvantage.
There is at the moment one other widely cited OODBMS benchmark called OO7 [10]. However, it is currently not as well updated as the pole position benchmark and therefore we decided to use the pole position benchmark.

2.4.2 Tracks

The pole position benchmark does not really simulate a formula one environment but uses its terminology for the different tasks. So a run on a track is an execution with set parameters on a certain type of task. During the first benchmark we discovered that at the moment it is enough if we only take the tests with simple data structures or queries into account, since we still lack a certain amount of performance at basic operations as one will see later on. Therefore we only used two tracks out of the currently implemented eight.
The Melbourne track basically writes a certain number of objects, which consists each of two longs and strings, to the database, reads them into main memory and deletes them. Table 2.1 shows the configuration values and table 2.2 explains the methods of this track. The Bahrain track works with the same objects as the Melbourne track but adds a few other operations to the test.

objects	number of objects to write into the database
commitinterval	number of object creations during one transaction

Table 2.1: Table showing the configuration variables for the Melbourne track

write	creates the objects
read	reads the objects into main memory
delete	empties the database

Table 2.2: Table showing the methods of the Melbourne track

2.4.3 Machine

All the test were run on the same machine which is described in table 2.5. Since we are simulating a instrumented version the query methods are not executed on the database but in main memory and therefore faster than an actual query into a database and this is why we will not compare query benchmarks in this thesis.

objects	number of objects to write into the database
selects	number of queries to perform
updates	number of updates to perform
commitinterval	number of object creations during one transaction

Table 2.3: Table showing the configuration variables for the Bahrain track

write	creates the objects
read	reads the objects into main memory
queryIndexedString	queries a for an object with an indexed string
queryIndexedInt	queries a for an object with an indexed integer
update	updates the object by changing one of the strings
delete	empties the database

Table 2.4: Table showing the methods of the Bahrain track

CPU	Intel(R) Core(TM)2 Quad CPU @ 2.4 GHz
RAM	1.98 GB
OS	Windows XP Professional SP3

Table 2.5: Table showing the metrics of the machine the benchmarks are run on

3 Storage Model

3.1 Model

The storage model proposed by [11] is a simple but efficient model for type description and storage. A type consist of three different base constructs:

- Base types - supported base types are string, integer, real, boolean, date, uri, clob and blob
- References - a reference to another object of any type
- Extents - supported extents are bag (unordered, duplicates), sequence(ordered, duplicates), set (unordered, no duplicates) and ranking (ordered, no duplicates)

A type is decomposed into its elements, called attributes, and described as a map of positions and base constructs. Figure 3.1 shows a simple person object consisting of a name and an age attribute.

3.2 API

Since this model is used as part of the OMS Avon system, there exists an API which can also be used on its own. We will have a look at some of the methods provided by this API which will be used later in this thesis.

3.2.1 Storage Management

A storage manager manages the different types of storages. At the moment the following storage implementations are available:

- In memory storage - a transient storage

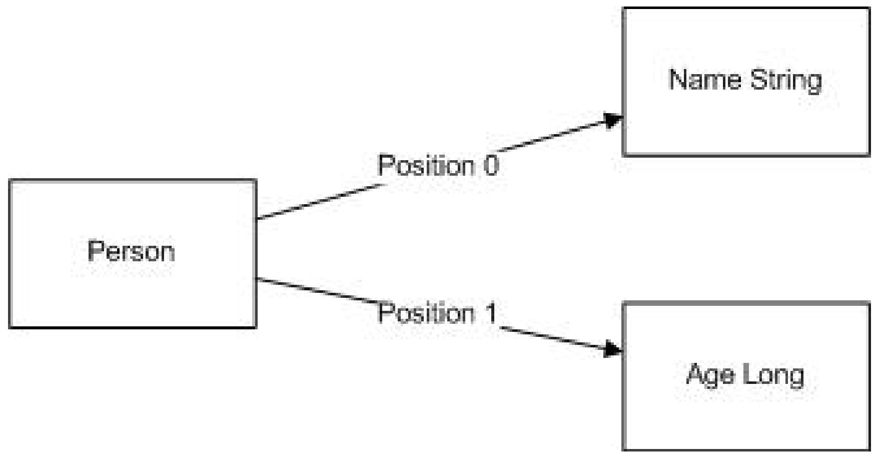

Figure 3.1: Example of a person with two attributes.

- Berkley DB old - an implementation based on Berkley DB
- Berkley DB new - an improved implementation over the other Berkley DB implementation
- db4o - an implementation using db4o as storage

A type of storage is selected by a configuration and a key string. Once a storage type is selected, one can create, delete and open a storage. Code listing 3.1 shows how a *Berkeley DB new* storage is created, opened, closed and deleted.

```
InternalConfiguration config = new BDBConfiguration();
StorageManager manager = StorageManagerFactory.getStorageManager(
    "bdb-new", config);

manager.createStorage("thesis");

Storage storage = manager.openStorage("thesis");

manager.closeStorage(storage);

manager.deleteStorage("thesis");
```

Listing 3.1: Creation retrieval and deletion of a storage.

3.2.2 Transactions

Most of the methods in the storage interface require a transaction handle. Transaction handles are issued by the storage. Every time a transaction handle is requested of the storage a new transaction will begin and is active until its transaction handle commits or rolls back. Code listing 3.2 shows the transaction handle methods.

```
TransactionHandle tx = storage.beginTransaction();

tx.commit();

tx.abort();
```

Listing 3.2: Retrieval of a transaction handle and its methods for commit and rollback.

3.2.3 Identifiers

One of the key components of the API are identifiers. They are used to identify types and objects as well. An identifier consists of a 128 bit uuid which will uniquely identify a component of the storage. Identifiers can take different roles depending on what they are representing, this role is assigned to them by the storage API. Identifiers are also used to store references to other objects or extents by storing an objects or extents identifier to an object reference attribute in the storage.

3.2.4 Type creation

To create a type, one has first to create the base types in the system. Code listing 3.3shows the creation of a base type.

```
Identifier longType = storage.createObject(tx);
storage.createBaseType(tx, longType, BuiltInType.INTEGER);
```

Listing 3.3: Creation of a base type.

Afterward these base types can be put together to an object type. For the example depicted in figure 3.1 this would look like code listing 3.4.

```
Identifier personType = storage.createObject(tx);
storage.createObjectType(tx, personType, new Identifier[]
       { stringType, longType });
```

Listing 3.4: Type creation.

3.2.5 Object creation

Objects are created by initialising an identifier like for type creation, but then dress it with a type. Objects can be dressed with any number of types. This concept of multi type dressing can be used to represent inheritance in the storage. Code listing 3.5 shows the creation and dressing of a person object.

```
Identifier person = storage.createObject(tx);
storage.dressObject(tx, person, personType);
```

Listing 3.5: Object creation.

3.2.6 Data Storing and Retrieval

To store and retrieve data, the following parameters are needed

- the object identifier
- the type identifier of the object
- the attribute position

These three values are kind of a primary key for an attribute value. Code listing 3.6 shows how an attribute is set and retrieved.

```
storage.setAttributeValue(tx, person, personType, position, "John");
name = (String) storage.getAttributeValue(tx, person,
        personType, position);
```

Listing 3.6: Sinlge value setting and retrieval.

Of course one can set and retrieve the whole attribute set of an object and its type as shown in code listing 3.7.

```
storage.setAttributeValues(tx, person, personType, new Object[]
        { name, age });

Object[] values = storage.getAttributeValues(tx, person,
        personType);
name = (String) values[0];
age = (Long) values[1];
```

Listing 3.7: Multi value setting and retrieval.

3.2.7 Example

Code listitng A.1 shows a persistent Java class which would implement our person example from section 3.1.

3.3 Orthogonality

We will have a look how far orthogonality is provided by the storage API. First we will have a look at static variables. Unfortunately static variables are not currently supported by the API. However, it does not require much change in the model and the API. The introduction of shared values would solve this problem.The API needs to be extended by one method which signature can be seen in code listing 3.8.

```
void makeAttributeShared(Identifier typeId, int position);
```

Listing 3.8: Method signature to enable static variable support.

There is another problem with integers and real values. The Java API matches the bases types real and integer with the object types Double and Long. An Integer has to be cast to a Long to before it can be stored using the API. Since this limitation, which can be solved by casting, still generates overhead for the developer or enhancer. It would be much more easier to add 32, 16 and 8 bit integers additional to the 64 integer. The same applies to floating point values.

4 Adding Transparency

4.1 Motivation

As stated earlier, transparency in the context of object persistence tries to reduce the efforts invested by a developer to persist objects. We will propose a solution which is nearly transparent except for two annotations, and will also show that in the context of transactional behavior and reachability transparency is not completely achievable without enormous efforts.

4.2 Enhancing

Speaking of enhancement in this context means to automatically change the code of a Java class, so that is stores and retrieves values from a database. Basically there are three points in the development and runtime process where enhancement could take place as shown in figure 4.1. Enhancement before compilation means that the written code is enhanced so the output is again readable Java code, whereas enhancement after compilation or before loading changes the Java byte code. Enhancement before compilation produces code which is still human readable and can be debugged whereas static byte code enhancing complicates debugging. Static enhancement on the other hand needs less complex enhancers since the layout of a compiled class file is strictly specified whereas Java source code has less strict rules. Compared to dynamic enhancement static enhancement generates less runtime overhead since the code does not have to be enhanced at runtime [12]. The big advantage of enhancement before loading over the two other approaches, is that it can be enabled or disabled before a program starts and therefore debugging of the semantics is possible without the overhead of a persistent storage - assuming, that the enhancer does not create any new bugs other than the ones already existing. Since there is already an approach with enhancement at load time for the OM model in Avon (of which the storage API is a core part) [13], we decided to use this approach again with the storage API.

Figure 4.1: Points where enhancement can be applied to Java code.

4.3 Java Byte code Instrumentation

Java byte code instrumentation [14] is a method to inspect and change the byte code of a class before it is loaded into the runtime of a program. There are some frameworks [12] [15] available to analyze and change Java byte code. However, these frameworks analyse the whole byte code of a class, whereas we only need information about the methods of a class, its private fields and its super class. Therefore we decided to write our own small analyzer. We will first discuss some difficulties and solutions with byte code instrumentation in regard of persistence in general before we go on to the solution we applied using the storage API.

4.3.1 Fields

Byte code instrumentation allows only insight into one class at a time and classes that have already been enhanced cannot easily be re-enhanced. Private fields are always set and read in the scope of the class, therefore no other classes have to be considered so that all the changes to persist them can be applied. Non private fields on the other hand belong to the interface of a class and therefore can be read and set by other classes. This means the enhancer has to know that a class is changing a persistent field and therefore firstly needs a list of all the persistent classes. This conflicts with circular references, like for example the persistent classes A and B both have a public reference of each other and both of them are changing the other reference at some point. While class B is loaded second it knows that it has to use the transparent code of A to changes A's reference of B; A which was loaded first did not know that B is a persistent class and would have to be re-enhanced. This little example illustrates, that in order to persist public fields, complex structures of references have to be hold by the enhancer and that quite a number of re-enhancing can go on before the whole code is enhanced. Therefore, non-private fields are always considered transient, and will not be persisted. This of course limits the orthogonality even more and is one of the major drawbacks of this approach.

4.3.2 References

Classes which are referenced by a persistent class have to be as well persistent and enhanced or the reference has to be transient, because, as seen in section 3.2, a reference in the storage model is the identifier of an object. This again limits the orthogonality of the system but is a common limitation in object oriented databases.

4.3.3 Collections

Collections are not quite as easy to persist transparently as references and fields since they cannot only be read and set but also can be changed by their interface methods, for example by adding a value to a list. There are two ways to cope with that problem. The first is based on a very deep inspection of the code to see where such methods are called and insert the appropriate byte code which handles the collection changes in the storage. A simpler

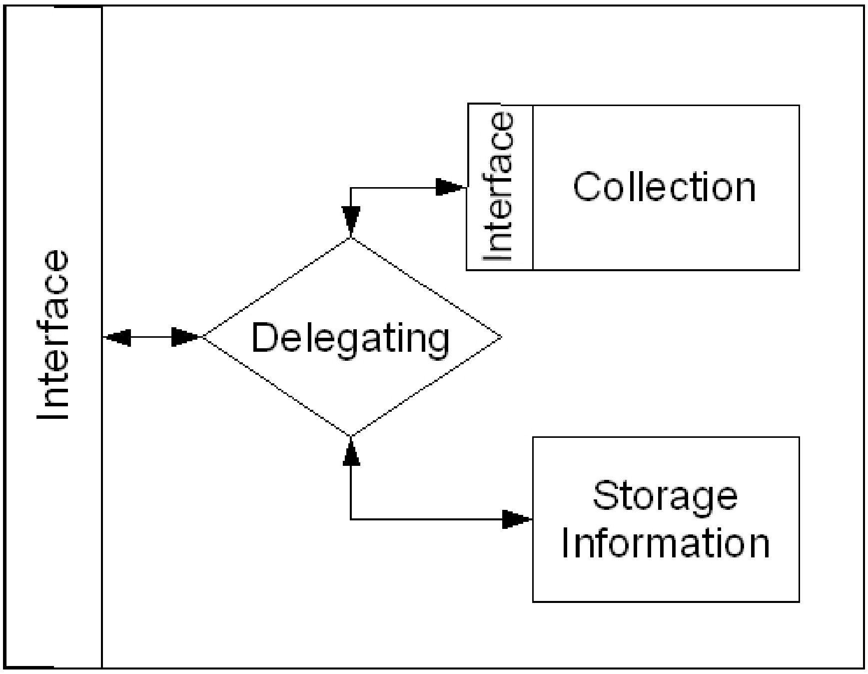

Figure 4.2: The components of a collection wrapper

but less efficient approach is to write wrappers to known Java collections. Figure 4.2 shows the concept of such a wrapper. It consists of the extent information used by the storage API and an in memory representation of the collection. The wrapper implements the delegator pattern, meaning that method calls are delegated to the in memory representation while simultaneously updating the representation in the storage. When a collection is set to a certain value, this value is wrapped before it is assigned to a field.

4.3.4 Inheritance

A persistent class extending another persistent class will use the identifier of its parent class and dress its persistent representation with the type identifier for this class. This assumes that all ancestor classes - except for Object the Java root class- are persistent. But what if a system or existing framework class is extended and this custom class has to be persistent. In this case one can mark a custom class as top of the inheritance structure for the perspective of the enhancer. This is done by adding the TopOfInheritance annotation to the class. If the enhancer framework finds this annotation, it will act like this class is inherited from Object which means, that it will generate an identifier for any objects created from this class.

4.3.5 Schema

Type information is stored in a schema which basically is a map which uses canonical class names as keys and type identifiers as values. This schema is stored at a fix identifier address in the database and is loaded at startup into a hash map which is globally available.

4.3.6 Enhancing Process

The diagram in figure 4.3 shows how a class is enhanced in general by our framework. First the byte code of a loaded class is decomposed by the Java class file standard [16], but only the constant pool, the fields and methods are fully decomposed whereas other information is skipped. Figure 4.4 shows in general detail which parts are used and which are skipped. The decomposition starts with the constant pool and checks if the MakePersistent annotation is present in the pool. If this check fails the framework does not continue, otherwise the fields and methods are decomposed as well. If this class is instrumented for the first time a type for this class is generated in the storage. Afterward a set of the private fields is generated, ignoring non-private fields, and the methods are scanned if they set or read any private fields and are marked for transformation. Then the constant pool is extended with the references to classes methods and static fields needed to invoke the storage API. Subsequently, the needed fields, methods and constructors are added to use the class with the persistent storage. We will have a closer look at the added elements later on. In the end the methods are transformed to make use of the added elements, how this is done we will see in the following sections.

4.3.7 Rules for Transforming Methods

With byte code instrumentation it is in general a good idea to change as little as possible in existing methods so that no unexpected errors will occur during runtime. From the experience with the bugs and crashes which came up during the implementation of the framework, we deduced a set of rules which are always good to be considered when enhancing byte code.

1. it is better to create a new method than to extend one
2. whenever possible insert code at the beginning or at the end of a method
3. make shared elements static

We will now see some explanation why we came up with this rules. Changing the byte code of an existing method is a delicate thing to do, especially if there are jumps present. Jumps in Java byte code use an offset to the next byte code instruction [17], so by inserting new instructions the existing offsets before need to be checked and possibly updated. At the beginning or end the stack is empty (except for any return values at the end), therefore there is no need to restore any old states except a return value which can be backed up and reloaded. Based on rule 1 and on the fact that adding a lot of code in every method which needs to be transformed could blow up a class dramatically, it is advisable to create new methods and invoke them than to inline them into the existing methods when working with byte code.
There are methods and fields which are independent of the layout of a class, like in our case the storage API instance, which can be implemented normally using Java code instead of byte code and made public and static as stated in rule 3. It facilitates the implementation of the framework since it reduces the amount of byte code one we had to write.

4.3.8 Implementation of the Byte Code Framework

Figure 4.5 shows the components of the framework. It consists of a constant pool component which handles the whole constant pool methods such as adding entries or getting the index of an entry as well as duplicate handling. One of the problems with the constant pool is the

index skipping for long and double values (defined in [16], section 4.4.5), which has to be handled accordingly.
The fields component is a container for the field informations and produces a list of the private fields. The methods component is a container for the methods using the code attribute to determine which of them have to be transformed, it also identifies constructors. Both, the field and method component, can also handle the insertion of new elements.

4.4 Implementation

Using our enhancement framework and the storage API, we started to enhance Java classes. There are two versions which are able to transparently persist Java objects. The second version is an improvement over the first one based on the insights gained from benchmarking.

4.4.1 First version

The first naive approach was to realise the methods described by the example in section3.2. This time we will have again a look at the person type and its enhanced equivalent. We also extend the person example so that it contains a reference and an extent value.
Code listing A.2 shows the unenhanced Java code.
To make the class in code listing A.2 persistent, first we have to add an identifier field to the type. This identifier is public so we eliminate the need for getters and setters. Code listing 4.1 shows the enhanced fields section.

```
private String name;
private Long age;
private Person spouse;
private List<Person> children;

public Identifier AID;
```

Listing 4.1: New field section.

Since we do no only want to create objects in the storage but want to load them as well we need a bypass constructor. In our case a bypass constructor takes an identifier value as seen in code listing 4.2. If a class extends a persistent class, the super bypass constructor has to be invoked as well.

```
public Person(Identifier AID) {
    this.AID = AID;
}
```

Listing 4.2: Bypass constructor.

The next thing we need to do is to extend the constructors so that they initialise the identifier and dress it. Following rule 2 there will be a dressing method added and a call to this method will be added to at the beginning each constructor shown in code listing 4.3. Dress methods in the first version have to be called second after the superconstructor is invoked since all the setters and getters need the object identifier. A class extending a persistent class does not create its own identifier but uses its parent identifier and dresses it with its storage type.

```
private void dress(){
    TransactionHandle tx = StorageContainer.storage.
        beginTransaction();
    AID= StorageContainer.storage.createObject(tx);
    StorageContainer.storage.dressObject(tx, AID,
                 StorageContainer.schema.get(this
                .getClass().getCanonicalName()));
    tx.commit();
}
```

Listing 4.3: Dress method.

After that we have to add the getter and setter methods which read and set values in the storage. First we will have a look at base types getters and setters which will be shown in code listing 4.4.

```
private void transparentSetName(String name) {
    TransactionHandle tx = StorageContainer.storage.
        beginTransaction();
    StorageContainer.storage
        .setAttributeValue(tx, AID, StorageContainer.schema.get(
        this.getClass().getCanonicalName()), 0, name);
    tx.commit();
}

private Object transparentGetName() {
    TransactionHandle tx = StorageContainer.storage.
        beginTransaction();
    Object result = StorageContainer.storage
        .getAttributeValue(tx, AID, StorageContainer.schema.get(
        this.getClass().getCanonicalName()), 0);
    tx.commit();
    return result;
}
```

Listing 4.4: Transparent getter and setter for base types.

A more complex thing are reference setters and getters because we have to consider null values. Code listing 4.5 shows the getter and setter for the spouse reference.
Collections just need a setter to wrap them as shown in code listing 4.6.
Now we need to transform the existing methods. And here we can ignore rule 1 because the size, stack input and output of the two instructions are identical as depicted in figure 4.6. therefore we can just replace them. After everything is put together we get the code in code listing A.3.

```java
private void transparentSetSpouse(Person spouse) {
    TransactionHandle tx = StorageContainer.storage.
        beginTransaction();
    if (spouse != null)
        StorageContainer.storage.setAttributeValue(tx, AID,
        StorageContainer.schema.get(this.getClass()
                .getCanonicalName()), 2, spouse.AID);
    else
        StorageContainer.storage.setAttributeValue(tx, AID,
        StorageContainer.schema.get(this.getClass()
                .getCanonicalName()), 2, null);
    tx.commit();
}

private Object transparentGetSpouse() {
    TransactionHandle tx = StorageContainer.storage.
        beginTransaction();
    Object result = StorageContainer.storage
            .getAttributeValue(tx, AID,
            StorageContainer.schema.get(this
            .getClass().getCanonicalName()), 2);
    tx.commit();
    if (result == null)
        return null;
    else
        return new Person((Identifier) result);
}
```

Listing 4.5: Transparent getter and setter for reference types.

4.4.2 Reachability

Objects are reached by a reference path starting from a root object which mostly is the one containing the main method. Knowing which of the objects have to be loaded from the database instead of being created, requires information about the state of an application (first run create, succeeding runs load for example). This kind of information has to be provided somehow and is therefore not transparent. We decided to use root objects which is a concept used by other systems as well for example by Versant. Root objects are like types stored in the schema of the storage with a string value as key. Code listing 4.7 shows how a root object is obtained by the storage container, the first parameter specifies the type of the object, the second parameter is the required string key for this root object, the third parameter is an array of types which is used to identify which constructor to invoke and parameter four are the values which have to be passed to said constructor. This concept of root objects in the database is also known as persistence by reachability.

4.4.3 Benchmark Result of First Version

After the first version was implemented we ran the benchmark to get some first results. We used the *Berkley DB old* provider implementation. As one can see clearly in figure 4.7 and

```
private void transparentSetChildren(List<Person> children) {
    children = new TransparentList<Person>(children, AID,
        List.class.getCanonicalName(),
        this.getClass().getCanonicalName(), 3,
        Person.class);
}
```

Listing 4.6: Transparent setter for extent types

```
Person mrJones = (Person) StorageContainer.root(Person.class,
    "personRoot", new Class[] { String.class, Long.class },
    new Object[] { "mr Jones", 45l });
```

Listing 4.7: Creation of root object

4.8 the enhanced version is much more slower than db4o. We believe this is due to a lot of database calls, for example all values have to be retrieved every time they are accessed, which probably use a lot of time. In a second version we will reduce the number of database calls and see how it compares to this version. As it can be seen we had to choose very small number of object so that the benchmark run would terminate in a practicable time. We will keep these numbers during this chapter so that different results are directly comparable.

4.4.4 Single Instances

There can be multiple instances of the same object present during the runtime, due to reference loading where just the identifier is retrieved and passed to the bypass constructor. In a scenario where you browse a lot of references this will lead to an enormous memory footprint. Therefore we introduce a container where identifiers and objects are matched. If a reference is loaded it is first checked that the object is not already loaded. For this container we use a weak hash map so that objects can be collected by the garbage collector even there is a identifier available for this object somewhere.

4.4.5 Second version

As benchmarking revealed, our first version of the instrumentation was quite a bit slower than the compared storages. To advance one could either improve the enhanced code or the storage underneath. Our first approach was to improve the enhanced code by reducing the number of database calls.

First we will have a look at the dress method called in the constructors. The API provides a second dress method which also takes values instead of only a type, so if we would not transform a constructor as we did and would set field values with the dress method we would reduce the calls to the database as seen in code listing 4.8, where the second dress method is used. The allInArray methods collects all the field values in an object array as shown in code listing A.4.

By the introduction of single instances we do not have to retrieve a field every time from the database as before, where we did not know if any of the instances of the same object applied any changes to its attributes. So we introduced a lazy loading for references and direct loading

```
private void dress() {
    TransactionHandle tx = StorageContainer.
        storage.beginTransaction();
    AID = StorageContainer.storage.createObject(tx);
        StorageContainer.storage.dressObject(tx, AID,
            StorageContainer.schema
            .get(this.getClass().getCanonicalName()),allInArray());
        tx.commit();
        Singletons.addSingleton(AID, this);
}
```

Listing 4.8: New dress method.

for extents and base values in the bypass constructor. Code listing 4.9 shows the new bypass constructor and the lazy loading. The allFromArray methods sets all the field values in an object as shown in code lisitng A.4.

```
public Person(Identifier AID) {
    this.AID = AID;
    spouseLoaded=false;
    allFromArray();
    Singletons.addSingleton(AID, this);
}

private void transparentGetSpouse() {
    if (!spouseLoaded && spouseId != null) {
        spouse=(Person) Singletons.
            getSingleton(spouseId, Person.class);
        spouseLoaded = true;
    }
}
```

Listing 4.9: New bypass constructor and lazy loading.

Every time a method sets a field the transparent setter is invoked. If for example a method sets a field in a loop it would generate a lot of database calls. By marking the changed variables and updating them at the end of the method we would reduce the number of database calls to one for every changed field in a method. This also applies for cunstructors, therefore the dress method is now called at the end of the constructor. Even more so if we would update all values in one call when more than one value has changed. Therefore we introduce a new update method as seen in code listing 4.10.
Overall we increased the memory footprint of a class and also generate sometimes calls which contain more information than necessary, but reduced the number of storage calls in the code by at least fifty percent. This number relies on the fact all the transparent getters are eliminated . Code listing A.4 shows the new enhanced person class.

```
private void update() {
    TransactionHandle tx = StorageContainer.
        storage.beginTransaction();
    StorageContainer.storage
            .setAttributeValues(tx, AID,
            StorageContainer.schema.get(this
            .getClass().getCanonicalName()), allInArray());
    tx.commit();
}
```

Listing 4.10: New update method.

4.4.6 Benchmark Results of Second Version

It is easy to recognise in figures 4.9 and 4.11 that we made huge improvements to the enhancing framework. Also it can be seen in figure 4.10 that reads now take longer. This can be explained by the introduced direct loading of base values. However, the results are still no match for db4o and we will try to improve them even more with the main focus on transaction management.

4.4.7 Transactions

As one can see transactions in the enhanced code are very short, mostly over one method. This generates quite some overhead and reduces the speed of the implementation. However, with transparent transaction such a fine granular approach is almost inevitable. After discussing different approaches including a transaction thread which commits after a certain amount of time and begins a new transaction or a method counter which commits after a certain number of method calls we decided, that these approaches were not really feasible because they meant considering thread safety and commits at points where they make no sense. After some consideration we came to the decision that transparency and transactions are not really compliant, because a transaction is normally a construct which contains a set of operations which in the context of the application belong together. It would require quite some analytical effort to deduce such sets and is not applicable with byte code instrumentation. Therefore we propose the following approach in which a global transaction manager can be used to manage transactions over the whole application. Basically it offers three methods:

- begin
- commit
- rollback

Using such a manager we are able to implement the commit interval - explained in table 2.1 and 2.3 - of the benchmark tracks and have a more accurate comparison than before.
Benchmark results show in figures 4.12 and 4.13 that we increased performance again and proofed that transactions generate quite some overhead. Comparing this version with db4o also showed that we now can compete with its implementation as seen in figures 4.14 and 4.15.

4.4.8 Hybrids

Since fully transparent solutions do not always meet the expectations and requirements of a system, the whole framework can be used as a hybrid with manually persistent code. This enables querying the storage which is a feature we did not consider so far because in a fully transparent environment it is assumed that your objects are in a kind of container where you implement your own query mechanism. In a fully transparent environment this requires however, that all the objects have to be loaded into main memory to be inspected, which is satisfied if the developer uses the Java collections or his own linked list for example. But sometimes one does not want to load all the objects into memory. For such cases it is possible to create a hybrid. Basically all the elements needed to query the storage are available to the developer. The storage object and schema are static and public and can therefore be used by any class. Identifier variables of an object can be retrieved by reflection; transactions are already non transparent so there is no problem to write queries manually as explained in [11].

4.4.9 Deletion

The aspect of deletion has so far been completely ignored due to limitations in the API. Our initial approach was to extend the finalize method which would delete an object inside the storage when the garbage collector deletes it. However, this might work in an environment where the application never stops or crashes since the main memory and and the storage are identical. But since there is almost no application which never stops, all the objects used during a runtime would in the end be collected by the garbage collector and therefore deleted. Furthermore is it possible that not all objects referencing an object are loaded into main memory and an object while rightfully collected by the garbage collector still is referenced in the database. Solving this problem requires reference counting in the storage and a weak delete method in the API where an object is only deleted if it his no longer referenced within the database and otherwise the deletion fails.

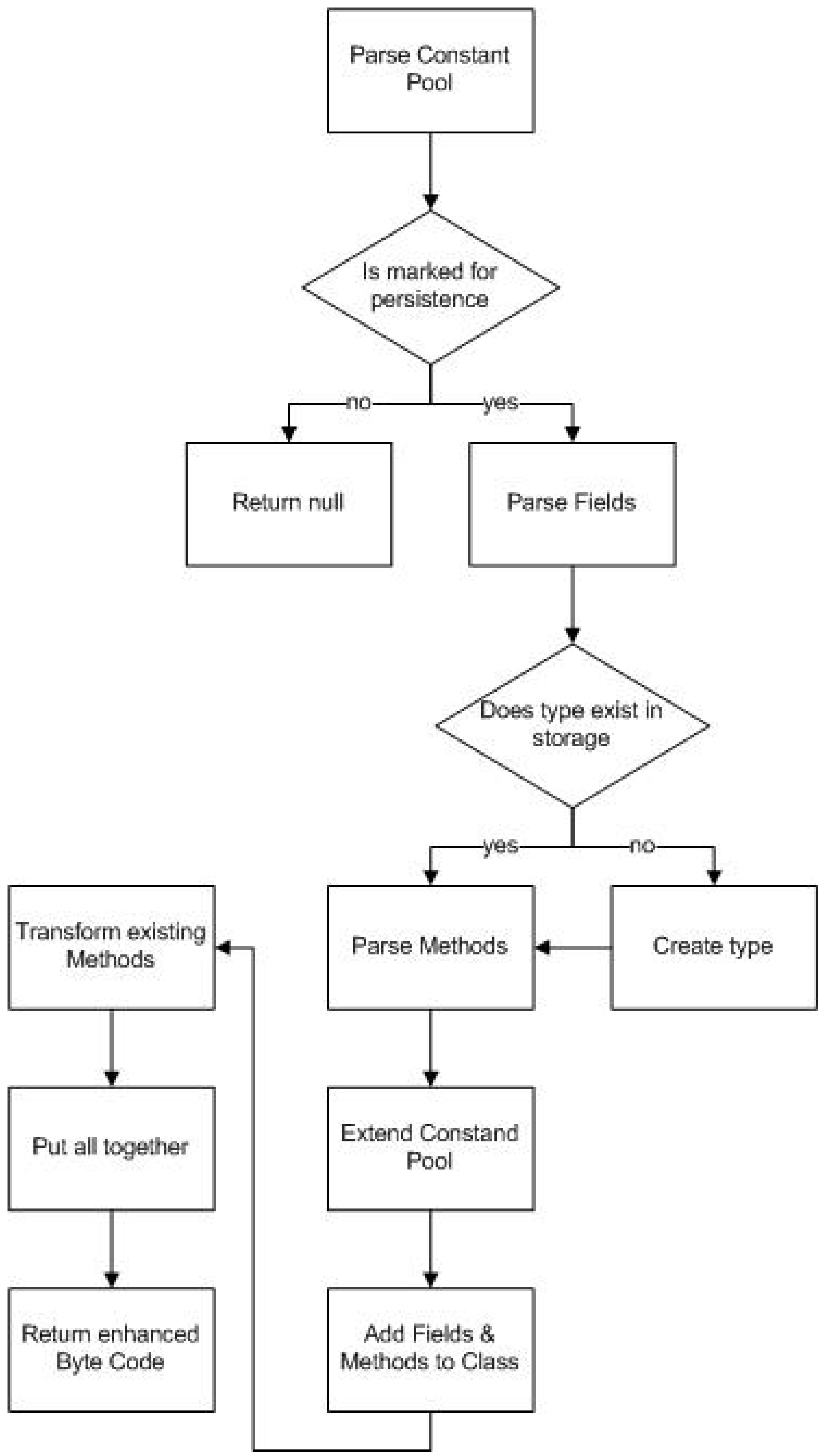

Figure 4.3: Diagram explaining the enhancement process

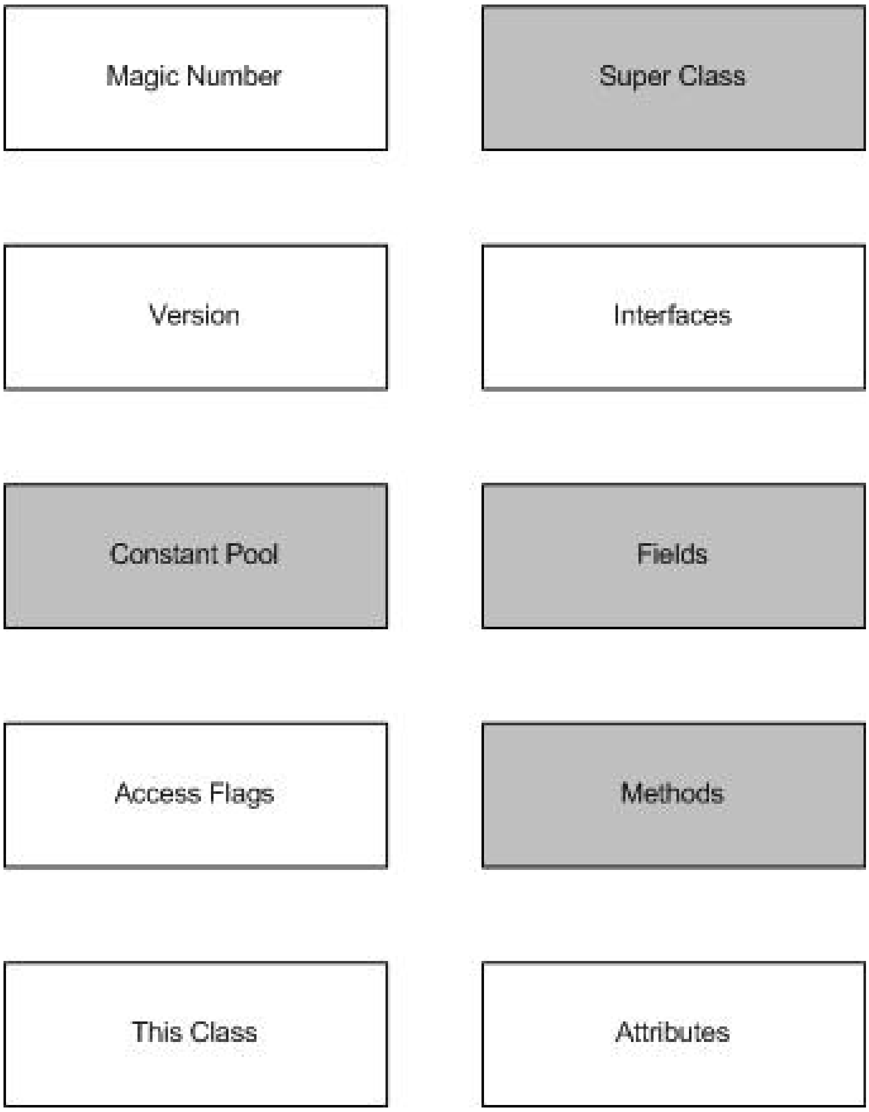

Figure 4.4: The gray boxes mark the sections of a class needed by the enhancing process

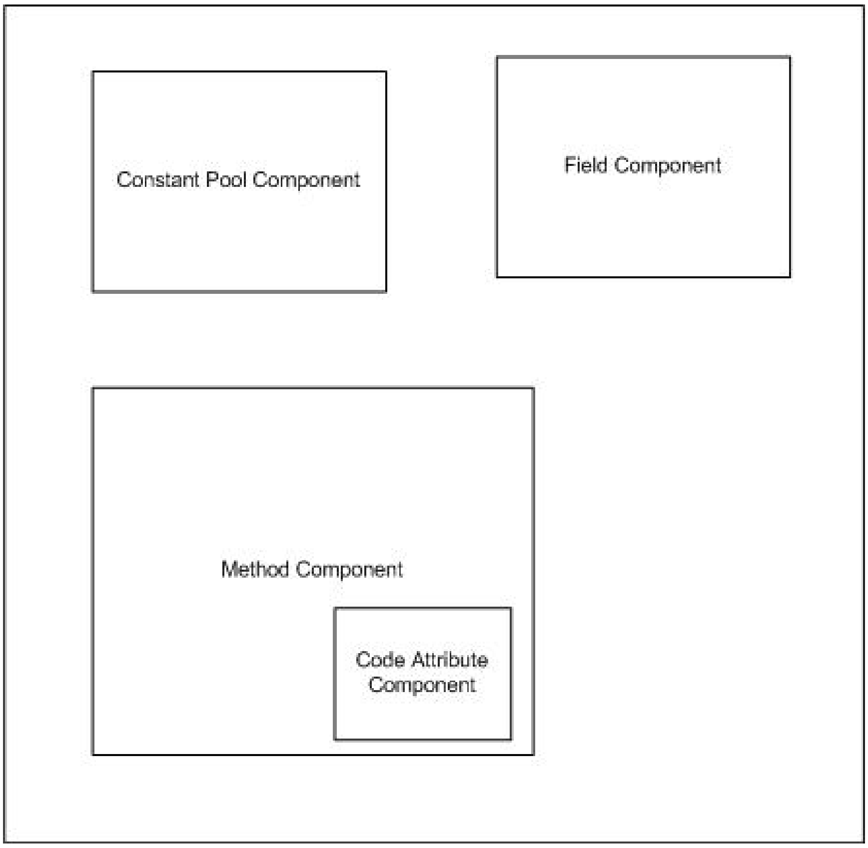

Figure 4.5: Showing the components of the enhancing framework

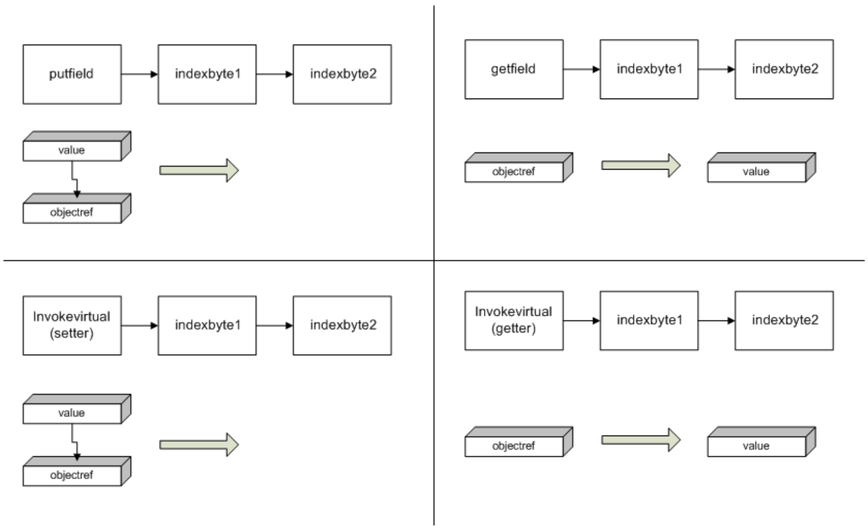

Figure 4.6: Instruction sizes and stack operations.

t [time in ms]	objects:100	objects:300	objects:1000
avon/first version	11157	44387	162673
db4o/8.0.145.14388	136	274	1044

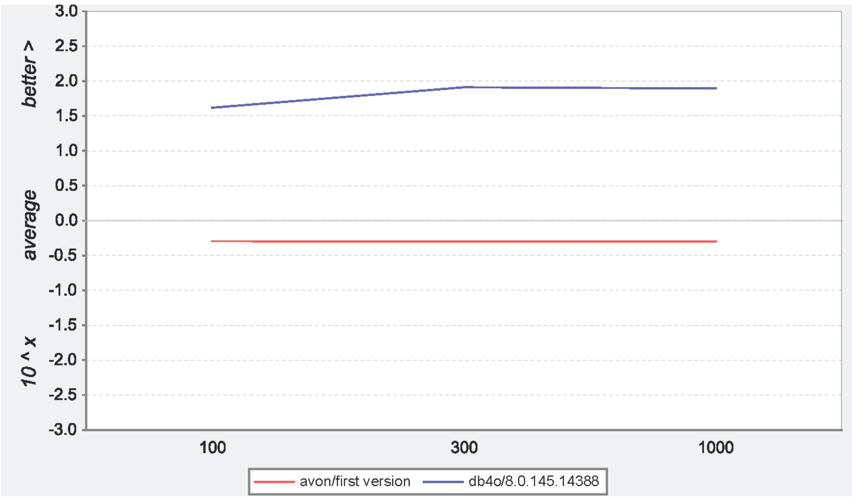

Figure 4.7: Melbourne track write, first version and db4o.

t [time in ms]	objects:300 selects:90 updates:80	objects:1000 selects:90 updates:80	objects:3000 selects:90 updates:80
avon/first version	1458	1955	1405
db4o/8.0.145.14388	118	120	156

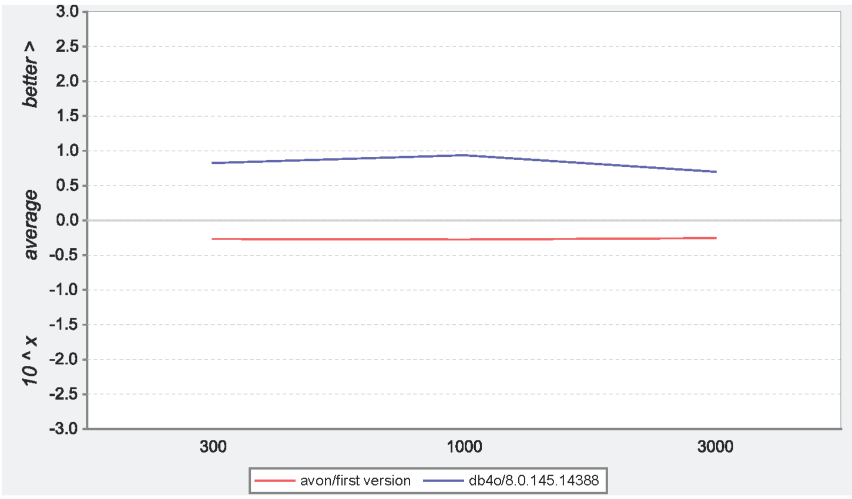

Figure 4.8: Bahrain track update first version and db4o.

t [time in ms]	objects:100	objects:300	objects:1000
avon/first version	12078	38622	137913
avon/second version	4163	12974	50897

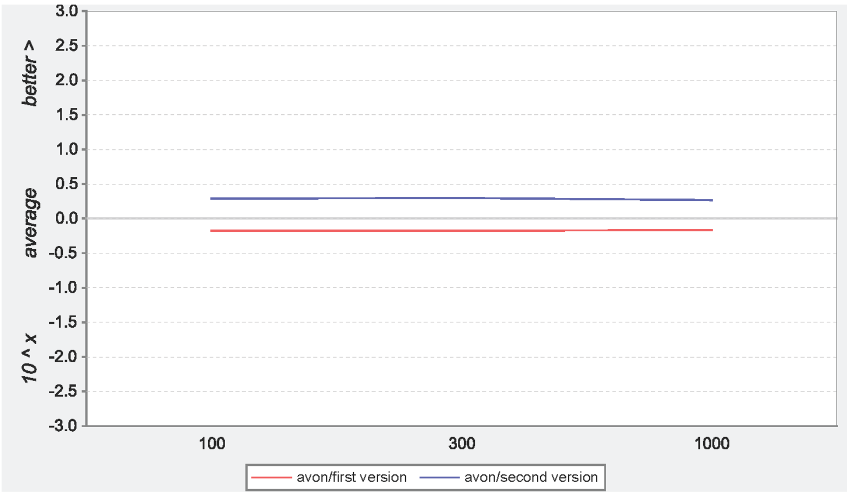

Figure 4.9: Melbourne track write, first and second version.

t [time in ms]	objects:100	objects:300	objects:1000
avon/first version	8	14	43
avon/second version	18	54	168

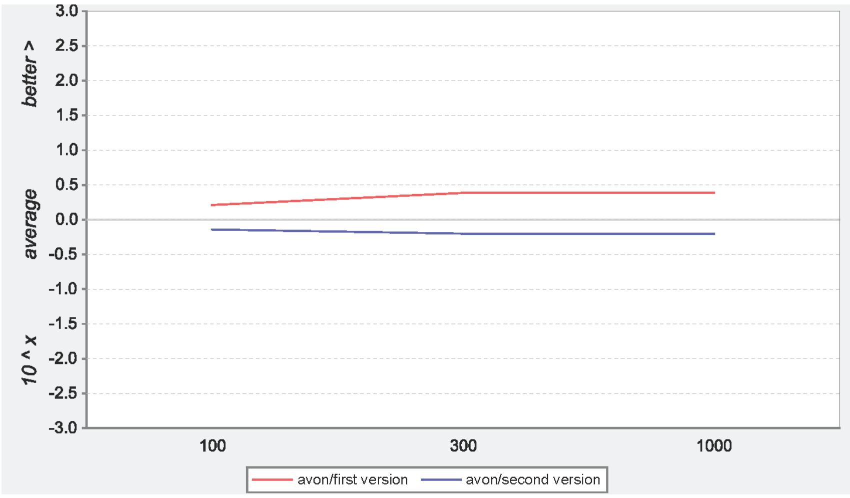

Figure 4.10: Melbourne track read, first and second version.

t [time in ms]	objects:300 selects:90 updates:80	objects:1000 selects:90 updates:80	objects:3000 selects:90 updates:80
avon/first version	4119	3925	4034
avon/second version	1600	882	1063

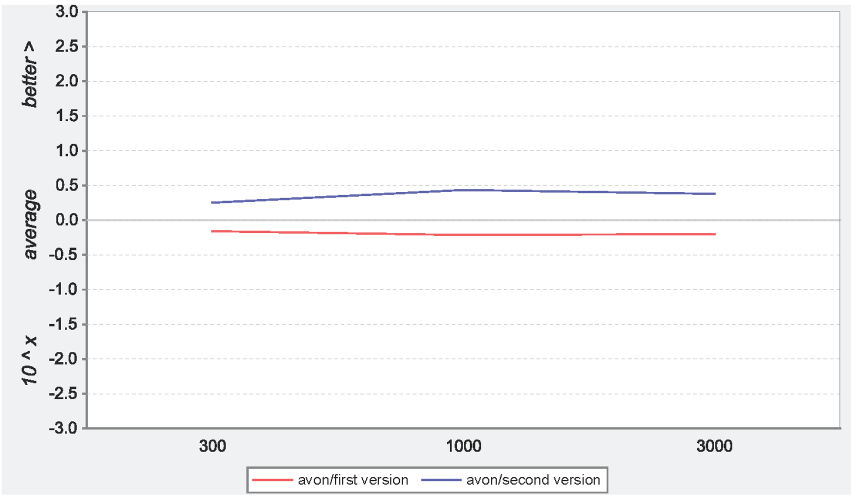

Figure 4.11: Bahrain track update, first and second version.

t [time in ms]	objects:100	objects:300	objects:1000
avon/transaction version	471	1007	6591
avon/second version	4261	12857	46119

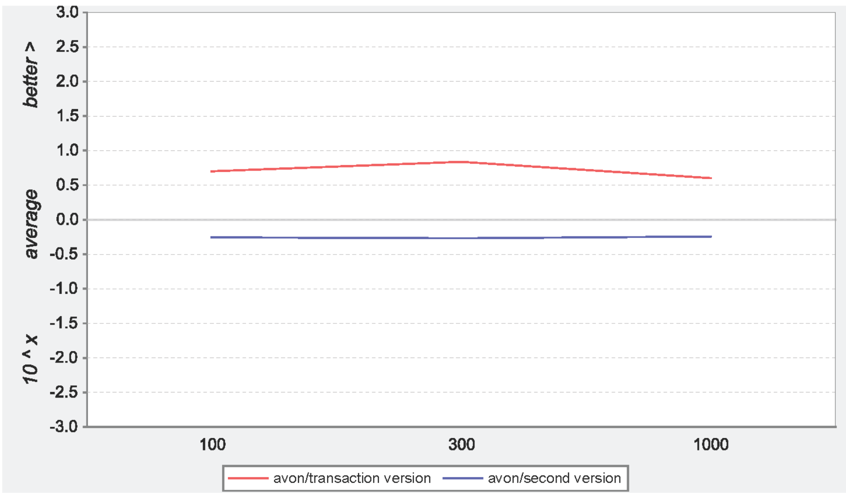

Figure 4.12: Melbourne track write, second and transaction version.

t [time in ms]	objects:300 selects:90 updates:80	objects:1000 selects:90 updates:80	objects:3000 selects:90 updates:80
avon/transaction version	45	58	47
avon/second version	1563	2310	2577

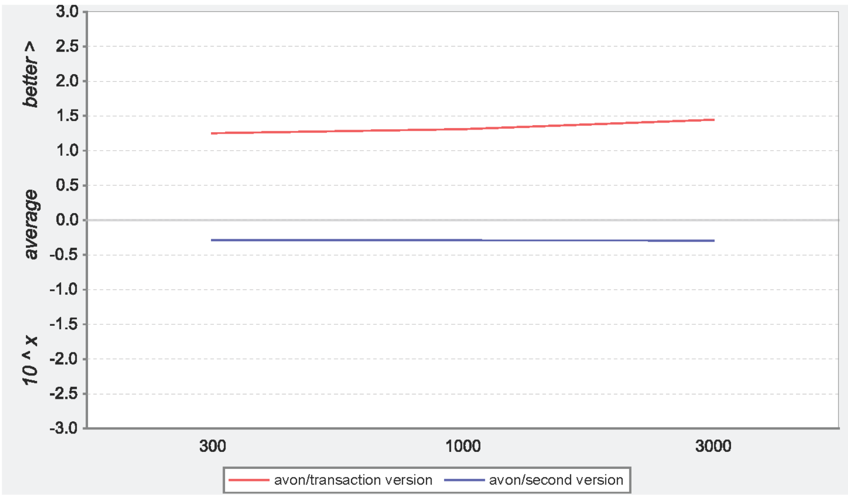

Figure 4.13: Bahrain track update, second and transaction version.

t [time in ms]	objects:100	objects:300	objects:1000
avon/transaction version	654	1750	8744
db4o/8.0.145.14388	925	2472	10233

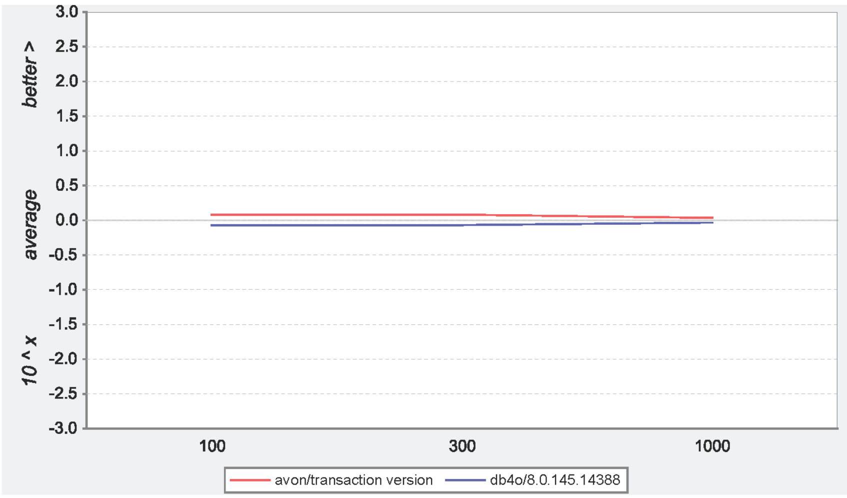

Figure 4.14: Melbourne track write, transaction version and db4o.

t [time in ms]	objects:300 selects:90 updates:80	objects:1000 selects:90 updates:80	objects:3000 selects:90 updates:80
avon/transaction version	56	59	38
db4o/8.0.145.14388	93	120	142

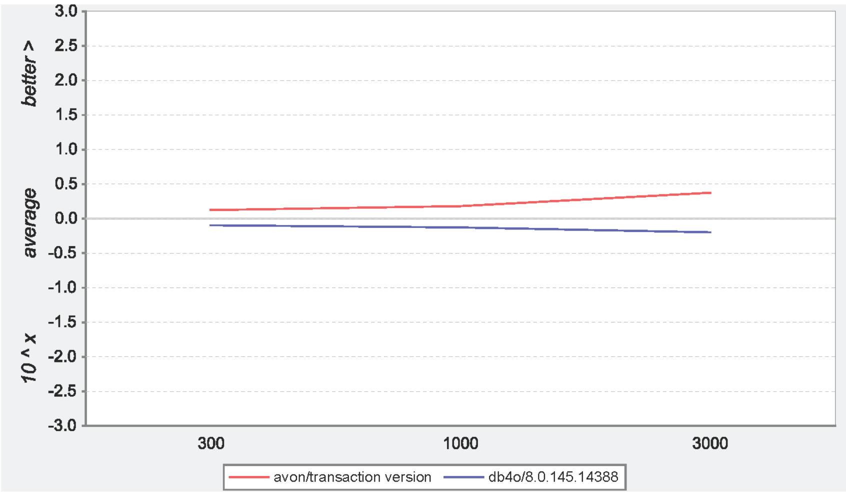

Figure 4.15: Bahrain track update, transaction version and db4o.

5 Storage Implementations

We have seen in section 3.2, that there are some different provider implementations available. We tried some other approaches and storage technologies with storage layers to experiment how the model can be adapted to other storage technologies and from which additional features of a provider we could benefit. These new providers implement only the necessary functionalities to run the benchmarks in order to evaluate the approaches.

5.1 MySQL

The main idea behind the MySQL [19] approach was to test the performance of the storage layer combined with a relational database, especially how it compares to Hibernate. Also it would be possible to use existing data by having an appropriate extract-transform-load method, a technique known from data warehousing.

5.1.1 Multi Table Version

This version creates a table for every base type in the model, as well as a table where one can describe types, and of course a table where object identifiers are stored. Figure 5.1 shows the model for the database. Values are stored in different tables according to their type. A value is identified by a type and an object identifier and a position as well. Identifiers have to be split up since MySQL does not support 128 bit integers. The high bits are marked by a s whereas the low bits are marked by an e at the end of the variable name, for example the object identifier oid would be split up into $<$oids, oide $>$. An object is stored in the objects table with its object identifier and type identifier. These two values are a combined primary key since an object can have multiple types. The definitions table simply holds the type identifier and the information which table to query for a certain value. So for example for a string value the value in the type column would be 'stringvalues'.
Benchmark results for this implementation show a huge improvement over the *Berkley DB old* implementation for object creation as seen in figure 5.2, However, it does not scale when

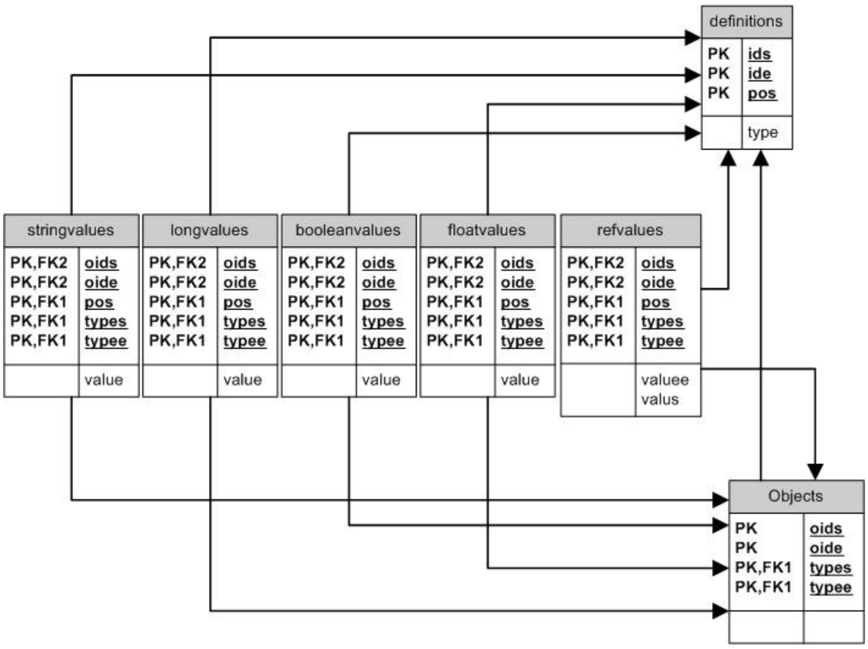

Figure 5.1: Model for the MySQL mutli table version.

it comes to updates as seen in figure 5.3. The update problem comes from the fact, that although we are using an index it is still more work intensive to find a record in the database as with BDB [24] where an update is more like an appended write into the database. We will later on have a look at a relational version with more promising update performance. Compared to Hibernate one can see that our implementation cannot compete with a highly optimised relational schema.

5.1.2 Single Table Version

The main idea behind this version was to test if we could increase speed of all operations by replacing the base value tables by one big table which would then be sparse, as shown on the ER model in figure 5.4. These changes are made so that the DBMS does not have to hold different pages for different tables in its buffer and therefore does not have to evict pages due to table changes and limited buffer size. As one can see in figures 5.5 and 5.6 this approach does not really improve writes or updates at all. This might result from the fact, that there are not so many page swaps in the buffer as one might suspect and the amount of data which has to be written into the database is now considerably larger than before.

5.1.3 Transaction Version

This version is an adaption of the multi table version, adapting to the transaction manager concept proposed in section 4.4.7. Here the database calls are stored in a batch job which is sent by the transaction handler to the database on its commit. This reduces all the calls during one transaction into one big call with multiple operations. This only applies for updates and writes whereas direct and lazy loads have to be performed immediately since the application requires the data. The singleton management of persistent objects prevents the application to

t [time in ms]	objects:100	objects:300	objects:1000
avon/bdb old version	1041	1890	9314
avon/mysql multi version	124	293	866
db4o/8.0.145.14388	909	3010	11057
Hibernate/MySQL-5.1	73	132	460

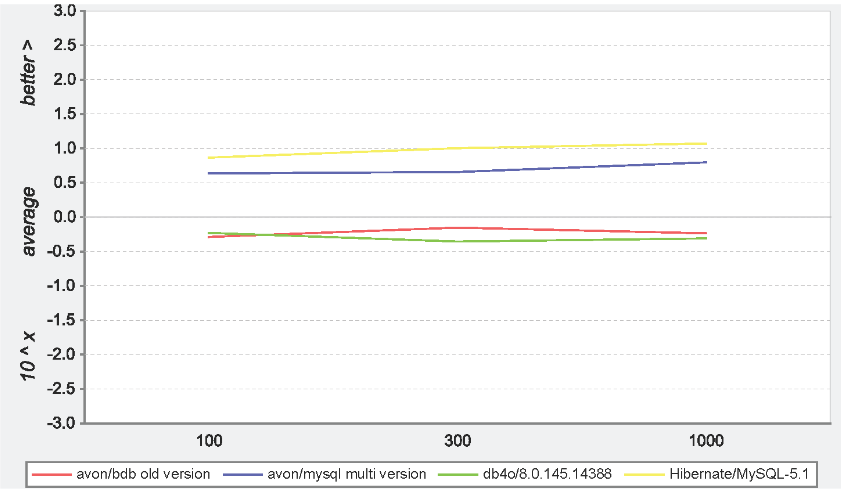

Figure 5.2: Melbourne track write, multi version, bdb old version, hibernate and db4o.

read old data since all the changes made in the database are also made in main memory as well.
The results in figures 5.7 and 5.8 show no significant improvements. This suggests, that database calls to MySQL are not that expensive.

5.1.4 Light Version

This version is identical to the multi table version but the interfaces are directly implemented removing all the code wrapped around the usual providers. This will test the overhead made by the storage layer framework.
Figure 5.9 shows that there is currently no significant overhead compared to all the additional checks and asserts performed by the API which guarantee consistency. However, with a faster storage this overhead becomes proportionally bigger.

5.2 Oracle

This version was used to test if higher update performance can be achieved if the storage provider has information about the logical or even physical address of the attributes values in the storage. We used Oracle [23] since it defines a ROWID value which is the logical address of a record in the database. An update supported with a ROWID can directly load the page where the data is loaded whereas a normal update has to scan an index or, even worse, a whole table. Basically this version is an adapted version of the multi table version but where

t [time in ms]	objects:300 selects:90 updates:80	objects:1000 selects:90 updates:250	objects:3000 selects:90 updates:800
avon/bdb old version	53	128	229
avon/mysql multi version	150	449	1410
db4o/8.0.145.14388	107	192	276
Hibernate/MySQL-5.1	52	139	444

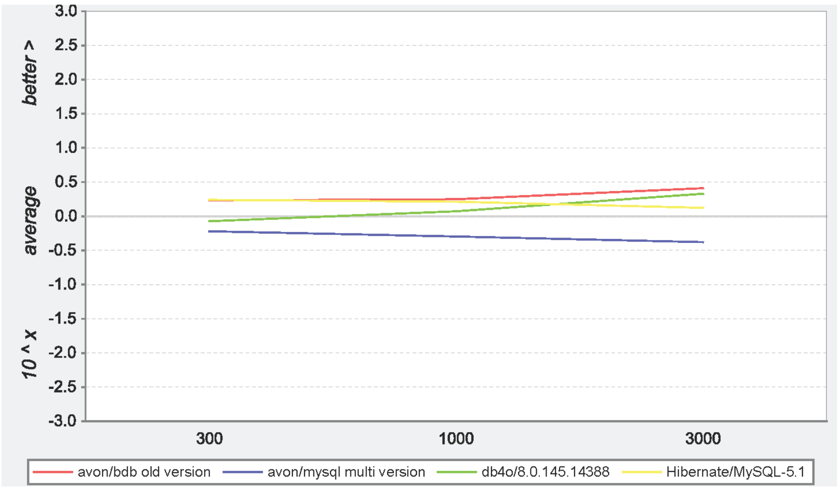

Figure 5.3: Bahrain track update, multi version, bdb old version, hibernate and db4o.

the ROWIDs of the different values of a type are stored in its identifier, and retrieved from there on updates. Writes and reads should be of the same speed, but we would expect that updates are considerably faster since the DBMS exactly knows where to change the data.
The benchmark result in figure 5.10 shows that a the version using the ROWIDs is faster than the Oracle version not using ROWIDs which confirms our assumptions. It has to be said that the Oracle instance we were using (10g express) was in no way configured or tuned. There could be better results with a configured instance.

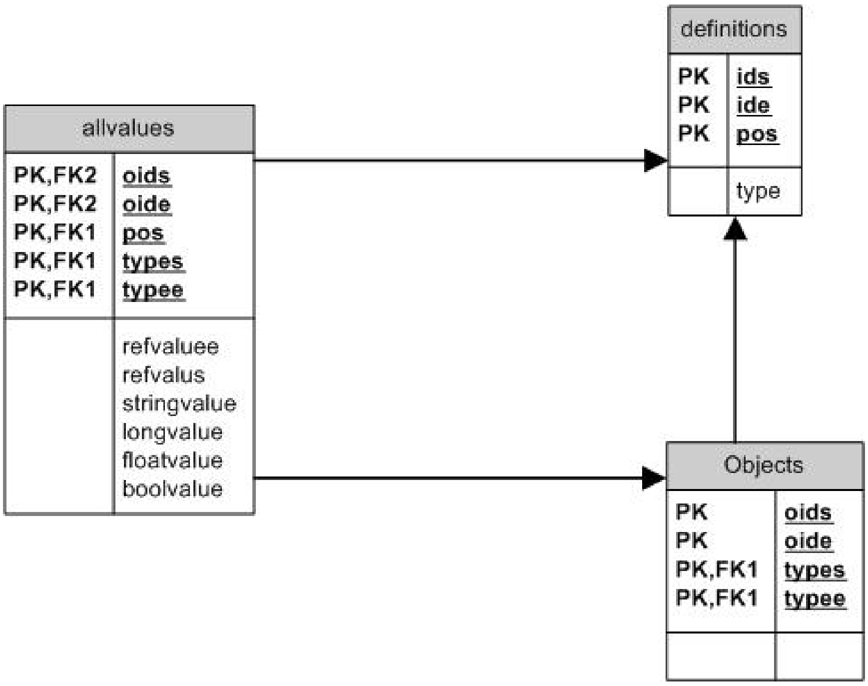

Figure 5.4: Model for the MySQL single table version.

t [time in ms]	objects:100	objects:300	objects:1000
avon/mysql multi version	129	279	796
avon/ mysql single version	85	229	739

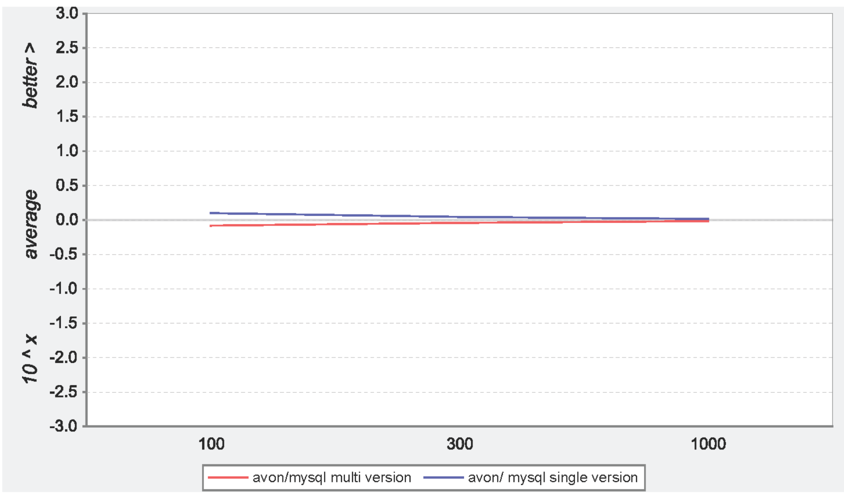

Figure 5.5: Melbourne track write, single and multi version.

t [time in ms]	objects:300 selects:90 updates:80	objects:1000 selects:90 updates:250	objects:3000 selects:90 updates:800
avon/mysql multi version	168	383	1289
avon/ mysql single version	154	467	1477

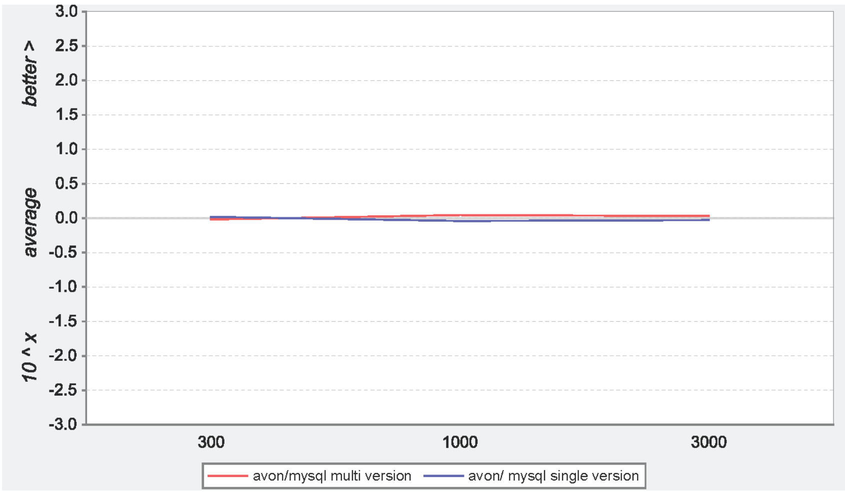

Figure 5.6: Bahrain track update, single and multi version.

t [time in ms]	objects:100	objects:300	objects:1000
avon/mysql multi version	145	318	930
avon/mysql transaction version	92	285	907

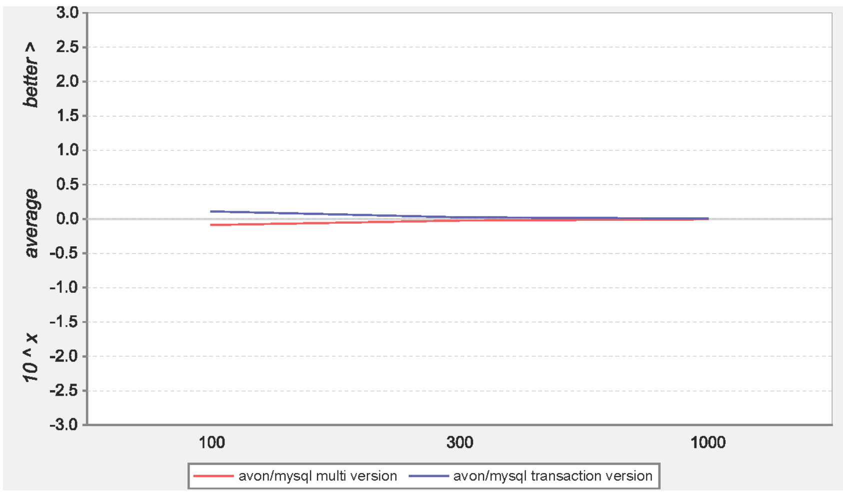

Figure 5.7: Melbourne track write, transaction and multi version.

t [time in ms]	objects:300 selects:90 updates:80	objects:1000 selects:90 updates:250	objects:3000 selects:90 updates:800
avon/mysql multi version	146	464	1436
avon/mysql transaction version	146	453	1424

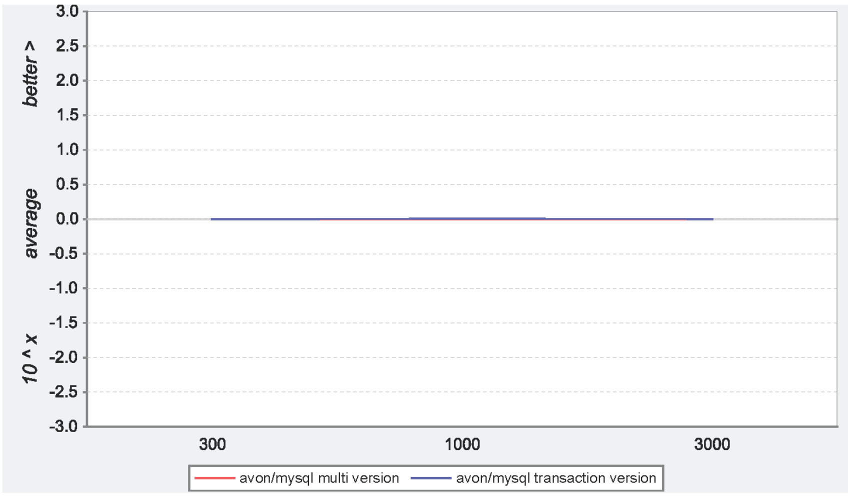

Figure 5.8: Bahrain track update, transaction and multi version.

t [time in ms]	objects:100	objects:300	objects:1000
avon/mysql multi version	130	275	792
avon/mysql light version	84	217	772

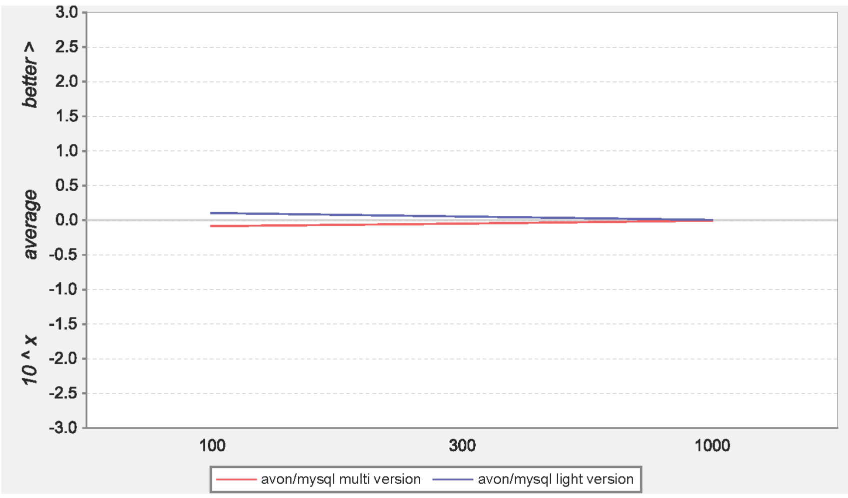

Figure 5.9: Melbourne track write, light and multi version.

t [time in ms]	objects:300 selects:90 updates:80	objects:1000 selects:90 updates:250	objects:3000 selects:90 updates:800
avon/rowid version	4096	12628	47071
avon/normal version	8283	27894	99891

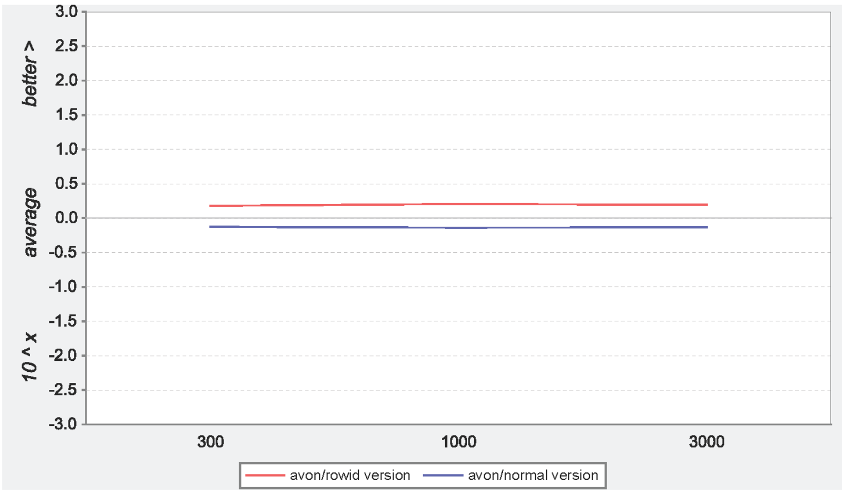

Figure 5.10: Bahrain track update, ROWID version and non ROWID version.

6 Conclusions

In this thesis we tried to integrate and evaluate a new storage provider which can be used to experiment with different kinds of storages and object models. The main focus of this thesis was to evaluate integration and performance.

6.1 Integration

We used dynamic byte code instrumentation to enhance Java code transparently (section 4.3). Although certain limitations resulted from this approach, mainly the problems with non private fields (section 3.3), the experience with byte code instrumentation in general was positive. We also showed that reachability (see section 4.4.2) and transactions (see section 4.4.7) are preferably handled not transparently for two reasons:

1. automatic analysis of such concepts is too complicated
2. they do not require much involvement from the developer

All in all we would say that integration of the storage provider can be maintained easily and is feasible using dynamic byte code instrumentation.

6.2 Performance

We have seen, that the storage providers using the right integration techniques containing a reduced number of database calls and non fine granular transactions, can achieve an better performances for writes and updates than our main competitor db4o while we have worst performance results for reads as shown in section 4.4.7. We could not compare features like queries and deletions due to the fact that deletion is not supported (section 4.4.9) and querying cannot be compared due to the transparent approach (see section 2.4.3). However, we were not able to get a better performance than an ORM approach which also might be a reason why developers still will use ORM frameworks in the future instead of an object oriented database as described in section 5.1. We also showed that updates can be improved if information about the location of the data is provided as seen in section 5.2.

6.3 Contributions

Since this is not an entirely theoretical thesis, there are some implemented products which come out of it. One is the instrumentation framework which is tightly tailored to the storage model trying to guarantee as little overhead as possible due to the enhancement at runtime. At the moment it uses the *Berkley DB old* provider. Also there are implementations for the Melbourne and Bahrain track using the API which are enhanced manually and not by instrumentation. This is done because the enhancer can only run with one storage provider, this manual implementations enable evaluation of different kinds of providers against each other. There are also stubs for the MySQL and Oracle storage providers which can be used to create a fully functional implementation.

7 Future Work

The results and conclusions from this thesis suggest a number of changes and further experiments concerning the storage provider.

7.1 Enhancer

The enhancer, while working fine with the benchmark tracks, needs to be tested in other situations to guarantee that it will not create any unexpected problems. Also it needs to be able to use any kind of storage provider defined by the developer.

7.2 Benchmark

While establishing good results for the Melbourne and Bahrain track other tracks of the pole position benchmark have to be implemented as well, in order to test the storage provider with more complicated data structures and tasks.

7.3 API and Model

The concept of shared variables has to be considered and eventually introduced. More importantly the limitations concerning the base types, described in section 3.3, should be removed. Also it has to be considered how the deletion mechanisms have to be adapted to support the byte code integration transparently.

7.4 Rollback

While the non-singleton implementation does always request values from the storage the singleton implementation keeps changes in main memory. This requires rollback mechanisms for this kind of implementation. The problems with singletons and rollback is that objects need to register to a transaction so that it knows which of the objects to rollback.

7.5 Relational Databases

It has to be discussed how far relational databases are used for storage providers. The advantage of relational databases lies in the fact that a lot of effort and money goes into their development, and they provide highly optimised features used by any kind of database like for example the transaction handling.

A Example Code Listings

A.1 Person Class Manually Made Persistent

```
public class Person {

    private Storage storage;
    private Identifier personType;
    private Identifier AID;

    public Person(Storage storage, Identifier personType) {
        this.storage = storage;
        this.personType = personType;
        TransactionHandle tx = storage.beginTransaction();
        AID = storage.createObject(tx);
        storage.dressObject(tx, AID, personType);
        tx.commit();
    }

    public String getName() {
        TransactionHandle tx = storage.beginTransaction();
        String result = (String) storage.getAttributeValue(tx, AID,
                 personType,0);
        tx.commit();
        return result;
    }

    public void setName(String name) {
        TransactionHandle tx = storage.beginTransaction();
        storage.setAttributeValue(tx, AID, personType, 0, name);
        tx.commit();
    }
```

```
    public Long getAge() {
        TransactionHandle tx = storage.beginTransaction();
        Long result = (Long) storage.getAttributeValue(tx, AID,
                personType, 1);
        tx.commit();
        return result;
    }

    public void setAge(Long age) {
        TransactionHandle tx = storage.beginTransaction();
        storage.setAttributeValue(tx, AID, personType, 1, age);
        tx.commit();
    }

}
```

Listing A.1: Person class manually made persistent.

A.2 Plain Person Class

```
public class Person {

    private String name;
    private Long age;
    private Person spouse;
    private List<Person> children;

    public Person() {
        this.name = "John Doe";
        this.age = 50l;
    }

    public Person(String name, Long age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Long getAge() {
        return age;
    }

    public void setAge(Long age) {
        this.age = age;
    }

    public Person getSpouse() {
        return spouse;
    }

    public void setSpouse(Person spouse) {
        this.spouse = spouse;
    }

    public List<Person> getChildren() {
        return children;
    }

    public void setChildren(List<Person> children) {
        this.children = children;
    }
```

```
    public void marry(Person spouse, String newName) {
        this.spouse = spouse;
        name = newName;
    }
}
```

Listing A.2: Plain person code.

A.3 Enhanced Person Class - First Version

```
public class Person {

    private String name;
    private Long age;
    private Person spouse;
    private List<Person> children;

    public Identifier AID;

    public Person() {
        dress();
        transparentSetName("John Doe");
        transparentSetAge(50l);
    }

    public Person(String name, Long age) {
        dress();
        transparentSetName(name);
        transparentSetAge(age);
    }

    public Person(Identifier AID) {
        this.AID = AID;
    }

    public String getName() {
        return (String) transparentGetName();
    }

    public void setName(String name) {
        transparentSetName(name);
    }

    public Long getAge() {
        return (Long) transparentGetAge();
    }

    public void setAge(Long age) {
        transparentSetAge(age);
    }

    public Person getSpouse() {
        return (Person) transparentGetSpouse();
    }

    public void setSpouse(Person spouse) {
        transparentSetSpouse(spouse);
    }
```

```java
    public List<Person> getChildren() {
        return children;
    }

    public void setChildren(List<Person> children) {
        transparentSetChildren(children);
    }

    public void marry(Person spouse, String newName) {
        this.spouse = spouse;
        name = newName;
    }

    private void transparentSetName(String name) {
        TransactionHandle tx = StorageContainer.
            storage.beginTransaction();
        StorageContainer.storage
                .setAttributeValue(tx, AID,
                StorageContainer.schema.get(this
                .getClass().getCanonicalName()), 0, name);
        tx.commit();
    }

    private void transparentSetAge(Long age) {
        TransactionHandle tx = StorageContainer.
            storage.beginTransaction();
        StorageContainer.storage
                .setAttributeValue(tx, AID,
                StorageContainer.schema.get(this
                .getClass().getCanonicalName()), 1, age);
        tx.commit();
    }

    private void transparentSetSpouse(Person spouse) {
        TransactionHandle tx = StorageContainer.
            storage.beginTransaction();
        if (spouse != null)
            StorageContainer.storage.setAttributeValue(tx, AID,
                StorageContainer.schema.get(this.getClass()
                .getCanonicalName()), 2, spouse.AID);
        else
            StorageContainer.storage.setAttributeValue(tx, AID,
                StorageContainer.schema.get(this.getClass()
                .getCanonicalName()), 2, null);
        tx.commit();
    }
```

```
    private void transparentSetChildren(List<Person> children) {
        children = new TransparentList<Person>(children, AID,
            List.class.getCanonicalName(),
            this.getClass().getCanonicalName(), 3, Person.class);
    }
    private Object transparentGetName() {
        TransactionHandle tx = StorageContainer.
            storage.beginTransaction();
        Object result = StorageContainer.storage
                .getAttributeValue(tx, AID,
                StorageContainer.schema.get(this
                .getClass().getCanonicalName()), 0);
        tx.commit();
        return result;
    }

    private Object transparentGetAge() {
        TransactionHandle tx = StorageContainer.
            storage.beginTransaction();
        Object result = StorageContainer.storage
                .getAttributeValue(tx, AID,
                StorageContainer.schema.get(this
                .getClass().getCanonicalName()), 1);
        tx.commit();
        return result;
    }

    private Object transparentGetSpouse() {
        TransactionHandle tx = StorageContainer.
            storage.beginTransaction();
        Object result = StorageContainer.storage
                .getAttributeValue(tx, AID,
                StorageContainer.schema.get(this
                .getClass().getCanonicalName()), 2);
        tx.commit();
        if (result == null)
            return null;
        else
            return new Person((Identifier) result);
    }

    private void dress(){
        TransactionHandle tx = StorageContainer.
            storage.beginTransaction();
        AID= StorageContainer.storage.createObject(tx);
        StorageContainer.storage.dressObject(tx, AID,
                StorageContainer.schema.get(this
                .getClass().getCanonicalName()));
        tx.commit();
    }
}
```

Listing A.3: Enhanced first verison of the person type.

A.4 Enhanced Person Class - Second Version

```
public class Person {

    private String name;
    private Long age;
    private Person spouse;
    private boolean spouseLoaded = true;
    private Identifier spouseId;

    private List<Person> children;

    public Identifier AID;

    public Person() {
        this.name = "John Doe";
        this.age = 50l;
        dress();
    }

    public Person(String name, Long age) {
        this.name = name;
        this.age = age;
        dress();
    }

    public Person(Identifier AID) {
        this.AID = AID;
        spouseLoaded = false;
        allFromArray();
        Singletons.addSingleton(AID, this);
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name=name;
        transparentSetName(name);
    }

    public Long getAge() {
        return age;
    }

    public void setAge(Long age) {
        this.age=age;
        transparentSetAge(age);
    }
```

```
    public Person getSpouse() {
        transparentGetSpouse();
        return spouse;
    }

    public void setSpouse(Person spouse) {
        this.spouse=spouse;
        transparentSetSpouse(spouse);
    }

    public List<Person> getChildren() {
        return children;
    }

    public void setChildren(List<Person> children) {
        transparentSetChildren(children);
    }

    public void marry(Person spouse, String newName) {
        this.spouse = spouse;
        name = newName;
        update();
    }

    private void update() {
        TransactionHandle tx = StorageContainer.storage
                .beginTransaction();
        StorageContainer.storage.setAttributeValues(tx,
                AID, StorageContainer.schema.get(this
                        .getClass().getCanonicalName()),
                allInArray());
        tx.commit();
    }

    private void transparentSetName(String name) {
        TransactionHandle tx = StorageContainer.storage
                .beginTransaction();
        StorageContainer.storage.setAttributeValue(tx, AID,
                StorageContainer.schema.get(this.getClass()
                        .getCanonicalName()), 0, name);
        tx.commit();
    }

    private void transparentSetAge(Long age) {
        TransactionHandle tx = StorageContainer.storage
                .beginTransaction();
        StorageContainer.storage.setAttributeValue(tx, AID,
                StorageContainer.schema.get(this.getClass()
                        .getCanonicalName()), 1, age);
        tx.commit();
    }
```

```
    private void transparentSetSpouse(Person spouse) {
        TransactionHandle tx = StorageContainer.storage
                .beginTransaction();
        if (spouse != null)
            StorageContainer.storage
                    .setAttributeValue(
                            tx,
                            AID,
                            StorageContainer.schema
                                    .get(this
                                            .getClass()
                                            .getCanonicalName()),
                            2, spouse.AID);
        else
            StorageContainer.storage
                    .setAttributeValue(
                            tx,
                            AID,
                            StorageContainer.schema
                                    .get(this
                                            .getClass()
                                            .getCanonicalName()),
                            2, null);
        tx.commit();
        spouseLoaded = true;
    }

    private void transparentSetChildren(
            List<Person> children) {
        children = new TransparentList<Person>(children,
                AID, List.class.getCanonicalName(), this
                        .getClass().getCanonicalName(), 3,
                Person.class);
    }

    private void transparentGetSpouse() {
        if (!spouseLoaded && spouseId != null) {
            spouse = (Person) Singletons.getSingleton(
                    spouseId, Person.class);
            spouseLoaded = true;
        }
    }
```

```
    private void dress() {
        TransactionHandle tx = StorageContainer.storage
                .beginTransaction();
        AID = StorageContainer.storage.createObject(tx);
        StorageContainer.storage.dressObject(tx, AID,
                StorageContainer.schema.get(this.getClass()
                        .getCanonicalName()), allInArray());
        tx.commit();
        Singletons.addSingleton(AID, this);
    }

    private Object[] allInArray() {
        Object[] result = new Object[4];
        result[0] = name;
        result[1] = age;
        if (spouse != null)
            result[2] = spouse.AID;
        else if (spouseLoaded && spouse == null)
            result[2] = null;
        else
            result[2] = spouseId;
        if (children == null)
            result[3] = null;
        else
            result[3] = ((TransparentList<?>) children).ext;
        return result;
    }

    private void allFromArray() {
        TransactionHandle tx = StorageContainer.storage
                .beginTransaction();
        Object[] result = StorageContainer.storage
                .getAttributeValues(tx, AID,
                        StorageContainer.schema.get(this
                                .getClass()
                                .getCanonicalName()));
        tx.commit();
        name = (String) result[0];
        age = (Long) result[1];
        spouseId = (Identifier) result[2];
        if (result[3] != null)
            children = new TransparentList<Person>(
                    (Identifier) result[3], Person.class);
    }
}
```

Listing A.4: Second version enhancement of person type.

List of Figures

List of Tables

Acknowledgments

I want to express my gratitude to my supervisor Christoph Zimmerli who always took the time to discuss issues and answer my questions. I was provided with great flexibility and freedom when working on this master thesis. Then I also would like to thank Prof. Moira C. Norrie for giving me the opportunity to write this thesis in the GlobIS group. The work was interesting and exciting.

Bibliography

[1] Rahul Biswas, Ed Ort. The Java Persistence API - A Simpler Programming Model for Entity Persistence, *Oracle Technology Network*, `http://www.oracle.com/technetwork/index.html`, 2006.

[2] Hibernate. `http://www.hibernate.org`.

[3] M. P. Atkinson , L. Dayns , M. J. Jordan , T. Printezis , S. Spence. An orthogonally persistent Java, *ACM SIGMOD Record*, v.25 n.4, p.68-75, 1996.

[4] Malcolm Atkinson , Ken Chisholm , Paul Cockshott. PS-algol: an algol with a persistent heap, *ACM SIGPLAN Notices*, v.17 n.7, 1982.

[5] OMS Avon. `http://maven.globis.ethz.ch/projects/avon`.

[6] Malcolm Atkinson , Ronald Morrison. Orthogonally persistent object systems, *The VLDB Journal The International Journal on Very Large Data Bases*, v.4 n.3, 1995.

[7] Mick J. Jordan , Malcolm P. Atkinson. Orthogonal Persistence for Java? - A Mid-term Report, *Proceedings of the 8th International Workshop on Persistent Object Systems (POS8) and Proceedings of the 3rd International Workshop on Persistence and Java (PJW3): Advances in Persistent Object Systems*, p.335-352, 1999.

[8] Moira C. Norrie. An Extended Entity-Relationship Approach to Data Management in Object-Oriented Systems, *Proceedings of the 12th International Conference on the Entity-Relationship Approach: Entity-Relationship Approach*, p.390-401, 1993.

[9] Pole position benchmark. `http://polepos.org/`.

[10] Michael J. Carey , David J. DeWitt , Jeffrey F. Naughton. The 007 Benchmark, *ACM SIGMOD Record*, v.22 n.2, p.12-21, 1993.

[11] Michael Grossniklaus, Alexandre de Spindler, Christoph Zimmerli and Moira C. Norrie. A Flexible Object Model and Algebra for Uniform Access to Object Databases, *In Proceedings of ICOODB 2010, 3rd International Conference on Object Databases*, p.56-70, 2010

[12] Walter Binder , Jarle Hulaas , Philippe Moret. Advanced Java bytecode instrumentation, *Proceedings of the 5th international symposium on Principles and practice of programming in Java*, 2007.

[13] Christoph Lins. Exlpoiting Dynamic Bytecode Instrumentation to Support transparent Persistence, Research Project in computer Science, ETH Zurich, 2009.

[14] Java Java.lang.instrument Package. http://download.oracle.com/Javase/6/docs/technotes/guides/instrumentation/index.html.

[15] Asm project. http://asm.ow2.org/.

[16] Tim Lindholm, Frank Yellin. The class File Format. *The Java Virtual Machine Specification*, chapter 4, 1999.

[17] Tim Lindholm, Frank Yellin. The Java Virtual Machine Instruction Set. *The Java Virtual Machine Specification*, chapter 6, 1999.

[18] Versant. http://www.versant.com.

[19] Mysql. http://www.mysql.com.

[20] db4o. http://www.db4o.com/.

[21] Objectstore. http://web.progress.com/en/objectstore/.

[22] Objectivity. http://www.objectivity.com/.

[23] Oracle Database 10g Express Edition. http://www.oracle.com/technetwork/database/express-edition/overview/index.html.

[24] Berkley DB. http://www.oracle.com/us/products/database/berkeley-db/index.html.