Performance Optimisation in an Object-Oriented Database Management System

Master Thesis

Christoph Zimmerli
<zimmerch@ethz.ch>

Prof. Dr. Moira C. Norrie
Alexandre de Spindler

Global Information Systems Group
Institute of Information Systems
Department of Computer Science
ETH Zurich

5th May 2009

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

globis

We should forget about small efficiencies, say about 97% of the time: Premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.

Donald Knuth

Abstract

OMS Avon is an object-oriented database management system developed in the Global Information Systems (GlobIS) Group at ETH Zurich. Besides serving as a proof-of-concept for the OM data model, the system's aimed uses range from rapid prototyping to teaching.

Adequate performance plays a crucial role in the acceptance process of a software system. Given the previously non-satisfying performance of OMS Avon, this thesis shows several aspects of performance analysis and optimisation in an object database management system. We first present a description of general techniques of performance analysis and optimisation. Because of its poor performance, the existing storage engine based on db4o is subjected to a redesign leading to a considerable performance increase. We also show the implementation of B^+-Trees as index structure for selection queries on attribute values and the associated increase in query performance.

Contents

1 Introduction

The wish for ”good performance” comes up with nearly one hundred percent certainty during every requirements elicitation process for a software system. While this aspect is certainly crucial for commercial products, it also applies to an academic environment. If we want people – and be it ”just” students who *have to* – to use an application, it should not be a waste of time in the sense that even the smallest use case takes a long time to complete. When trying to push our implementation as proof-of-concept for a proposed model, having acceptable performance is also desirable. Otherwise, people will discard our idea as ”nice, but obviously not really feasible” at best.

Unfortunately, most of the above actually applied to OMS Avon before this thesis – at least when using the db4o-based storage implementation. Therefore the original task for this thesis was to review the existing implementation of the db4o-based storage layer. After analysing its weaknesses regarding performance, a better approach should be derived and implemented.

Having completed this original task in half of the time given for this thesis, a second topic needed to be chosen. Going along with the first optimisation-related part, implementing additional indexing in OMS Avon was the task of choice for the second half.

1.1 Document Structure

After this introduction, Chapter 2 will provide an overview over some principles of performance analysis and optimisation. Chapter 3 then introduces OMS Avon and its underlying model. The work on the original topic of optimising the db4o-based storage implementation is presented in Chapter 4. Following that is Chapter 5 about the second unit of work concerning performance optimisation in OMS Avon through indexing. Finally, an overview over the accomplishements and ideas for further development based on the work associated with this thesis will be given in Chapter 6.

The source code presented in listings in this document will be written in Java. This is due to the fact that OMS Avon is implemented in Java and hence most of the programming work associated with this thesis was done in Java.

2

Analysing the Performance of a Complex System

Before actually starting the process of optimisation, it is crucial to know which parts of the system should be optimised. While it may well be possible to gain some performance in about every part of a complex system, there usually exist hot-spots where optimisation shows a much greater influence than in other places.

Such a hot-spot is for instance a part of code that takes very long to execute. Let us look at an example: Suppose we have a program that executes three methods. Two of those methods take one second to execute each. The third one runs for 20 seconds. This leads to a total execution time of 22 seconds for our small program. Let us assume that optimising any of those three methods has no impact on the performance of the other two methods. In this setting it is obvious that one will benefit the most from optimising the third method. If we would work on the two one-second-calls, the best total execution time we could reach would be 20 seconds (reducing those two calls to zero seconds each). Were we however able to reduce the third method to zero, we would get an execution time of only two seconds!

The assumption made in the example above, that optimising one method has no impact on other methods, often does not hold in complex systems. In particular since one of the principles of object-oriented software construction is reusing software components. This implies that it would make sense to asses how often certain pieces of code are executed when running the program. Optimising a method, that is executed a hundred times during a run of the program, to run one second faster will result in a total execution time that is one hundred seconds smaller than before. Suppose we would instead optimise another method's execution time by two seconds. We could say that this optimisation is twice as good as the other one that brought only an improvement of one second per execution. But if this second method is only executed three times during one run of our program, these two seconds per execution will

only result in a total speed-up of six seconds per program execution, instead of the hundred seconds that could be achieved by optimising the first method.

These two small examples above show that the real hot-spots will be found in places that *a*) take long to execute and *b*) are executed very often. The following sections of this chapter describe methods and tools to identify such parts in a complex system. We will look at whole methods in order to identify hot-spots. This is no limitation on granularity. After all, a method could always be split up into several or merged together with others to adjust granularity.

2.1 Counting Method Executions

As stated above, the number of times a method is executed is one metric that is useful for identifying hot-spots in a program. So let us have a look at how we can count the number of times a method is executed.
Suppose we have a method `someMethod` as shown in Listing 2.1.

```
class Something {

   public void someMethod() {
      executeSomething();
   }

}
```

Listing 2.1: Class `Something` containing a simple method `someMethod`.

Counting the number of executions of `someMethod` is simple and straight forward, as Listing 2.2 shows.

```
class Something {

   public void someMethod() {
      logger.logCall("someMethod");
      executeSomething();
   }

}
```

Listing 2.2: Logging calls to `someMethod`.

The call of `logger.logCall("someMethod")` is supposed to increment the number of executions we keep for `someMethod`. This can be accomplished in numerous ways such as keeping a map of method name and execution count, writing a log file for later examination or by storing the information in a database.

Using the idea just shown entails adding code to each method whose execution count one is interested in. This may be acceptable, if one only wants the numbers for a small amount of methods. If the number of calls of many methods is of interest, or one does not want to alter the existing code to get those numbers, this idea is not an option.
The remedy to this dilemma can be found in proxy classes. The idea of a proxy class is to provide the same interface, as the class whose method executions should be counted. Listing 2.3 shows such a proxy class `SomethingProxy` that acts as a proxy for the class `Something`. Proxies that record information about method calls, like `SomethingProxy`, are generally called *tracing proxies*.

```
class SomethingProxy {

   private something = new Something();

   public void someMethod() {
      logger.logCall("someMethod");
      something.someMethod();
   }

}
```

Listing 2.3: A proxy class that logs a call and then forwards it.

As one can see from this small example, using proxies requires you to write such a proxy class for each class whose methods you would like to trace. However, as we are working with Java, we do not have to do that all by hand, because Java offers us something called *Java Dynamic Proxy* [1]. It enables us to create proxies for any class by providing all the interfaces the proxy should implement, as well as an `InvocationHandler`. The latter allows us to specify the actions to be carried out whenever a method is executed on the proxy object. Implementing an `InvocationHandler` that handles method calls by always logging the call before actually executing the method can be done as shown in Listing 2.4.

```
class LoggingInvocationHandler
   implements InvocationHandler {

   private Object target;     // the object being proxied

   public Object invoke(Object proxy, Method method,
      Object[] args) throws Throwable {
      logger.logCall(method.getName());
      return method.invoke(target, args);
   }
}
```

Listing 2.4: An `InvocationHandler` for logging method calls before execution.

[1] `http://java.sun.com/javase/6/docs/technotes/guides/reflection/proxy.html`

2.2 Measuring Method Execution Time

The second metric useful for identifying hot-spots mentioned in the introduction of this chapter is method execution time. Going back to our simple example in Listing 2.1, we could measure the execution time of `someMethod` by starting a timer at the beginning of that method, and stopping it at the end of the method (see Listing 2.5).

```
class Something {

   public void someMethod() {
      long start = System.nanoTime();
      executeSomething();
      long end = System.nanoTime();
      long executionTime = end - start;
      // store executionTime for later analysis
   }

}
```

Listing 2.5: Measuring a method's execution time.

The results of such time measurements are not always very useful. First, the execution time of a method is not constant. It may vary from execution to execution. Second, methods sometimes take less time to execute than can be measured. This can occur due to the fact that `System.nanoTime()` actually cannot guarantee nanosecond accuracy, as this method relies on a timer offered by the operating system. Therefore it can happen that we measure an execution time of zero nanoseconds. It is however clear that executing this very method one billion times will probably not take zero nanoseconds as well, as one could expect by naively multiplying the time for one execution with one billion.

```
class SomethingTest {
   private static final int EXECUTION_COUNT = 1000;

   public void testSomeMethod() {
      long start = System.nanoTime();
      for(int i = 0; i < EXECUTION_COUNT; i++) {
         someMethod();
      }
      long end = System.nanoTime();
      long totalExecutionTime = end - start;
      long averageExecutionTime =
         executionTime / EXECUTION_COUNT;
      // store averageExecutionTime for later analysis
   }
}
```

Listing 2.6: Executing a method several times and calculating the average execution time.

In order to overcome those two problems, it is a good idea to execute the method under test a fixed number of times and measure the total time taken for all the executions. We then get the average time for one method execution simply by dividing the total execution time by the number of executions. This process is shown in Listing 2.6.

If all methods to be timed run long enough to get execution times greater than zero nanoseconds and each method is executed often enough during the whole execution of the program, the idea of proxies, and in particular the *Java Dynamic Proxy* introduced in Section 2.1, can be applied for measuring method execution time as well. We simply implement an invocation handler similar to what is shown in Listing 2.7.

```
class LoggingInvocationHandler
   implements InvocationHandler {

   private Object target;     // the object being proxied

   public Object invoke(Object proxy, Method method,
      Object[] args) throws Throwable {
      long start = System.nanoTime();
      Object result =  method.invoke(this.target, args);
      long end = System.nanoTime();
      long executionTime = end - start;
      // store executionTime for later analysis
      return result;
   }

}
```

Listing 2.7: An `InvocationHandler` that measures the execution time of a method.

2.3 Profiling

As one can imagine, we are not the first people interested in performance analysis of software. Therefore several tools exist to facilitate the job at hand. The action of analysing performance in software is often called *profiling*. Programs to help with profiling are known as *profilers*. A profiler automatically collects the data discussed in Section 2.1 and Section 2.2. Sometimes it is necessary to compile the program with a special tool in order for the profiler to work. For instance, to analyse a program with *gprof*, a free profiler that is part of the GNU Binutils [2], the program must either be compiled or linked with an additional option set on the compiler or linker.
For profilers on the Java platform, the Java Virtual Machine offers the *JVM Tools Interface (JVMTI)*. This interface allows profilers to register for events like method calls, class loading or unloading and entering or leaving threads.

[2] `http://www.gnu.org/software/binutils/`

During the work of this thesis we used *JProbe* [3] to analyse the performance of a Java program. JProbe automatically combines the two metrics of counting method executions and measuring the time per execution. The visual presentation of the results then shows the *critical path*, which identifies the profiled application's main path of execution.

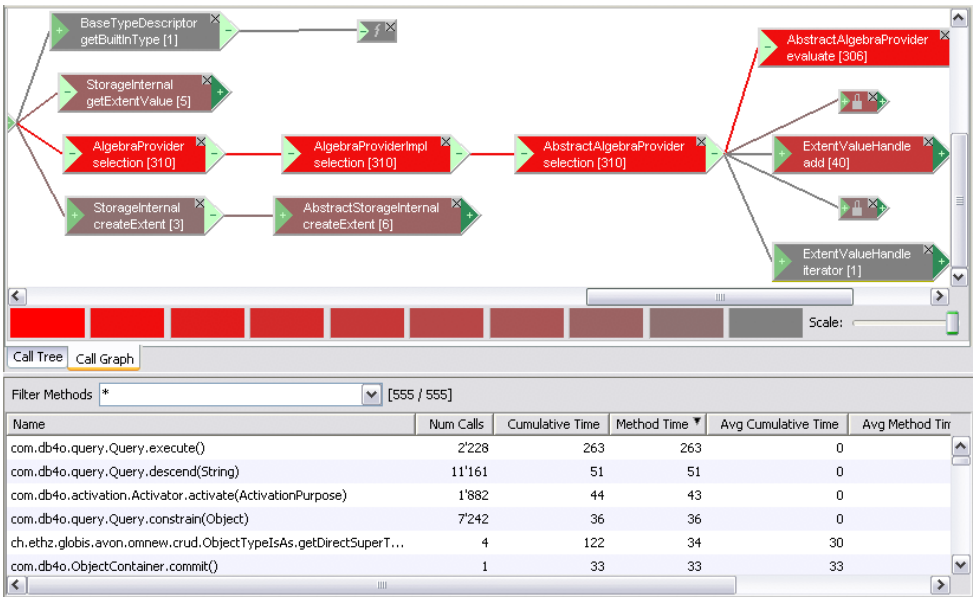

Figure 2.1: A screen-shot of JProbe's performance analysis.

Figure 2.1 shows a small example of such results of a performance analysis. In the bottom part of the screen, JProbe shows a table containing the number of calls and the total execution time per method. Above that table there is the visual representation of the program's call graph. Each node represents a method. The lines between the nodes show where the method was called from (incoming lines from the left) and what other methods it called (outgoing lines to the right).
The boxes representing methods are coloured by cumulative time. Cumulative time is the sum of time spent in a method and all its callees during the program's whole execution. Methods with highest cumulative time are shown in red. The farther away a method is from the highest value, the more grey its colour is. This colour-representation allows a quick visual assessment of the critical path and therefore the parts of the program that should be optimised first.

[3] http://www.quest.com/jprobe/

3 OMS Avon

OMS Avon is the latest incarnation in the history of implementations of the OM Datamodel. Previous versions were OMS Pro, OMS/Java and eOMS, all built by the Global Information Systems group at ETH Zurich [1].

This chapter first presents the base for Avon, the OM data model, in Section 3.1. It then discusses Avon's architecture in Section 3.2. Finally, Section 3.3 will provide a look at db4o, the key ingredient of the storage implementation discussed in the upcoming Chapter 4.

3.1 The OM Data Model

The OM data model is an extended Entity-Relationship model for object-oriented data management. One of its key features is the distinction between typing and classification by using a two-layer model. It also treats collections and binary associations as first-order concepts. Together, these features allow expressing multiple inheritance, multiple instantiation and multiple classification.

In order to represent the two-level model and other features, OM has its own graphical representation. An example of such a diagram is shown in Figure 3.1. It depicts a part of OM's core meta-model and will be discussed in parts in the following.

As in other object-oriented models, a type also declares a set of attributes and methods in OM. Each object has at least one object type defining its behaviour. In Figure 3.1 these object types are represented by the collection `ObjectTypes`, whose entries are of type `objectType`.

[1] `http://www.globis.ethz.ch/research/oms`

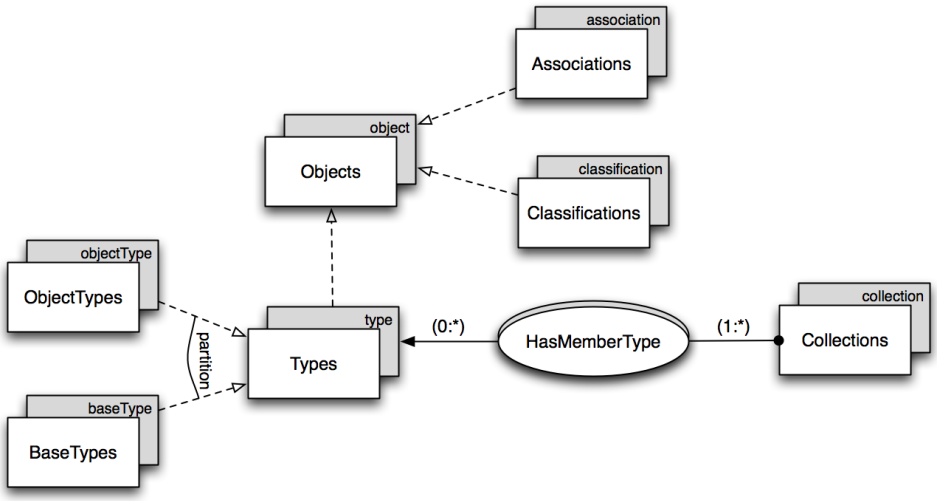

Figure 3.1: Graphical representation of a part of OM's core meta-model.

Object types can form inheritance hierarchies like in most other object-oriented models. But in contrast to many of these models, OM supports *multiple inheritance* by allowing a type to have several supertypes.
Besides object types, there exist also base types, illustrated by the collection `BaseTypes` with members of type `baseType`. Examples for such base types are strings, integers or booleans.

Throughout its lifetime, an object can gain new and lose existing types. The operation of gaining a type is called *dress*, while losing a type is known as *strip*. This concept of having several types at the same time is known as *multiple instantiation*.

Classification allows grouping objects semantically with the help of the types defined in the type layer. This grouping of objects is achieved by the use of collections. While being in a collection, an object participates in the role defined by the collection's membertype.
Allowing an object to be member of several collections at the same time enables it participating in multiple roles at the same time, which is known as *multiple classification*.

3.1.1 Collections

The graphical representation of a collection is a rectangular box. Figure 3.1 consists mainly of collections and relations between them. The text on the foreground rectangle is the collection's name. The grey-shaded rectangle in the background shows the type the objects in a collection have. It is known as the collection's *membertype*. As root element in the figure, we have the collection `Objects` containing members of type `object`, i.e. all objects.

OM also enables us to express constraints over collections. Figure 3.1 depicts on the left that the collection `Types` has two subcollections: `ObjectTypes` and `BaseTypes`. Between those two there exists the *partition* constraint. This means that a type can either

be an object type or a base type, but not both at the same time. Besides *partition*, OM also supports *intersection*, *disjoint* and *cover*.

OM distinguishes four different kinds of collections – one for each combination of allowing duplicates and knowing an order:

- *Set*: No duplicates allowed, entries have no order.
- *Bag*: Duplicates allowed, entries have no order.
- *Sequence*: Duplicates allowed, entries are ordered.
- *Ranking*: No duplicates allowed, entries are ordered.

3.1.2 Associations

Associations are used to relate objects in one collection to objects in another collection. In Figure 3.1, such a relation is shown between the collections `Collections` and `Types`. Represented by the oval shape and carrying its name `HasMemberType`, this association models the concept of collection membertypes introduced above.

An association also defines cardinality constraints. In our example, the `(1:*)` states that each collection must have at least one membertype. On the other hand, a type can be the membertype of zero or more collections, indicated by `(0:*)`.

3.2 Avon Architecture

OMS Avon is implemented in Java. So far, it consists of two major layers as depicted in Figure 3.2. Each of these layers will be introduced in the following sections.

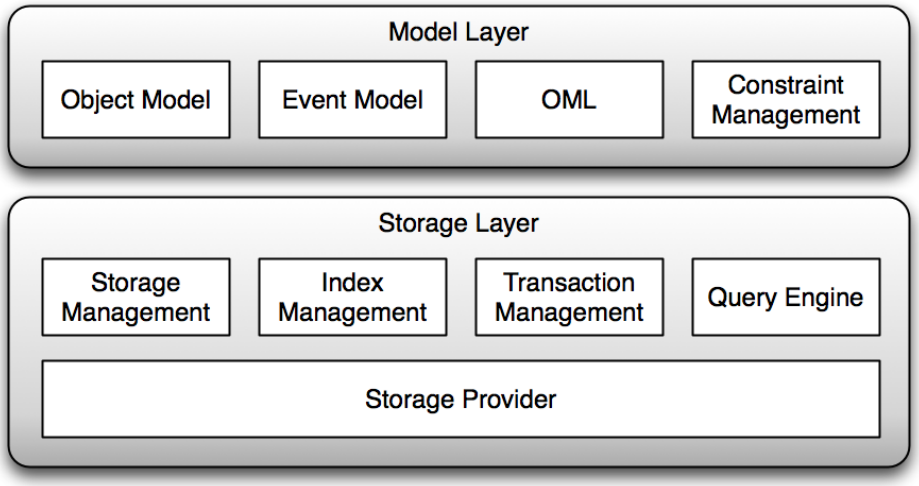

Figure 3.2: High-level view of Avon's architecture.

3.2.1 The Model Layer

The model layer implements the OM data model discussed in Section 3.1. The implementation provides a generic Java abstraction for OM objects called `OMObject`. It allows the creation and manipulation of such OM concepts as object types, collections and associations. Java, the implementation language, itself does not support all OM concepts, such as multiple inheritance and multiple instantiation. Therefore, `OMObject` provides this functionality, for instance through its methods `dress` and `strip`.

It is the model layer's responsibility to manage the constraints. This includes for instance checking that at least one of an object's types corresponds to the membertype when adding an object to a collection. OML, a modeling and query language for the data stored in Avon is also part of this layer.

3.2.2 The Storage Layer

This layer's task consists in mapping the data of the model layer to an internal representation that can then be stored persistently. Details of this internal representation will be discussed in Section 4.2. The storage layer also manages indices and transactions and provides a query engine.

In order to make the data persistent, the storage provider employs an external persistence provider. Because of its modular design, it is possible to implement different storage providers, each using a different persistence provider. As of writing this thesis, there exist three such implementations:

- *In-Memory*
  This implementation just maps the objects to plain Java data structures. It does not provide any actual persistence – if the application shuts down, all data is lost.
  It is, obviously, the fastest performing implementation.
- *Berkeley DB*
  For using Berkeley DB as a persistence provider, this implementation maps the Java object structure to a relational model.
- *db4o*
  Because db4o is an object database (see Section 3.3), this implementation can store and retrieve normal Java objects to and from the database.

3.3 db4o

db4o [2] is an open source object database that runs on both Java and .NET. It supports storing and retrieving arbitrary object graphs including collections.

3.3.1 Query Interfaces

To retrieve the stored data, db4o supports four query interfaces.

[2] `http://www.db4o.com`

Query-By-Example (QBE) In order to use QBE, the user provides a template object. db4o will then return all of the objects in the database which match all non-default field values in the template. This is done via reflection on all of the fields and building a query expression where all non-default-value fields are combined with *and* expressions. Default values are what the programming language initialises new variables with, for example `0` for integers and `null` for references.
Using QBE, one can only express simple queries (no *or* or *not* clauses and the like) and not constrain on values equal to the default-value for that field type.

Native Queries (NQ) Native Queries are promoted by db4o as the main query interface and to be absolutely type-safe, compile-time checked and refactorable. These claims all hold because NQ are – as the name implies – native to the programming language.
Creating a NQ means implementing a `Predicate` containing a `match` method with boolean return value. Inside this method, any program code – including method calls – can be executed to decide, whether a given candidate object matches this query. All objects in the database, for which the `match` method returns `true` will be part of the query result.

LINQ (Language Integrated Query, .NET only) For LINQ the same properties hold, as for NQ. However, LINQ is not an invention of db4o, but a feature of .NET itself. db4o simply provides the right interface to make its data accessible through LINQ.

SODA (Simple Object Database Access) SODA allows the user to build query graphs by hand using constraints like $greater, smaller, equal, and, or$ and methods for string comparison. It is also possible to sort the results of a SODA query. All these constraints have to be made to object field values – SODA does not allow method calls as part of the query. As the names of these fields are provided as strings, SODA queries are not as easily refactorable as NQ.
All other forms of queries will internally be translated to SODA for execution. db4o knows several optimisation possibilities for this translation. But because it cannot optimise all types of queries (yet), certain queries will be executed in a slow unoptimised manner.

3.3.2 Transparent Persistence and Activation

One of db4o's goals is to hide the use of the database as much as possible from the application logic. *Transparent Persistence (TP)* is a key element for this purpose. It allows registering an object once with the database through a call to db4o's `store` method. Any further changes to the object will then be observed and automatically persisted by the database. TP is very useful for the implementation of Avon's storage layer using db4o as persistence provider. It enables creating an object at the storage layer, registering it with the database and then passing it to the upper layer for modification, without actively having to care for occurring changes on the object.
In order to enable TP, the object's class has to implement db4o's `Activatable` interface. This can either be done by hand or through instrumentation – either at compile time or when the class is loaded by the class loader.

The same `Activatable` interface plays also a role in hiding the fact that an object has been retrieved from the database. As mentioned before, db4o can store arbitrary object graphs. When querying for the root object of a large graph, db4o does not know, whether the user is just interested in this root object, or if he will traverse the whole graph from there. Every object the user wishes to access has to be instantiated – or activated – by db4o. Because this process is quite time-consuming, it would be a bad idea to always fully activate query results. The user can instruct db4o to always activate results to a fixed depth and / or handle activation by hand. However, this process of activating by hand does not adhere to the goal of hiding the use of a database.

Using *Transparent Activation (TA)*, the activation process can be hidden from the user. TA will activate query results on demand and as lazily as possible. Making a class TA-aware, and therefore enabling lazily activating it, is also accomplished by implementing `Activatable` (TP includes TA).
Retrieving a query result, db4o will activate the graph until it encounters an object implementing `Activatable`. This object will be lazily activated on demand. This means that TA-unaware objects will also work in a TA-enabled environment without additional care for their activation.

4

Accelerating Avon's db4o-based Storage Module

As discussed in Chapter 3, OMS Avon has the ability to use different implementations of the storage layer. Such an implementation – with the exception of the pure in-memory implementation – uses an external persistence provider. One of the existing implementations uses db4o [1] to persist the application's data.

While being able to pass all the necessary test-cases correctly, the db4o-based implementation was not actually considered usable because of its poor performance.
For instance, a small test-case called `OMPerformanceTest`, that operates on Avon's model-layer, took nearly seven seconds to execute. For comparison: The same test would run in one tenth of a second using the in-memory implementation.

While it is obviously unrealistic to expect the same execution time as with in-memory when using a DBMS [2] to store and retrieve data, it was desirable to increase the performance to a level that could be considered usable for real-life use-cases.

The upcoming sections of this chapter will describe the analysis and decisions made during the task of improving the performance of the db4o-based storage module and show the resulting performance gains.
During this, the db4o-based implementation's state before the work associated with this thesis will be called *db4o-old*. In contrast, the optimised version resulting from the described work will be called *db4o-awesome*.

[1] `http://www.db4o.com/`
[2] Database Management System

4.1 The `OMPerformanceTest`

Before discussing the optimisation process itself, let us first have a look at what will serve as a benchmark for measuring improvements.
The `OMPerformanceTest` is a small use case that consists of 29 subtasks. For each of these subtasks the test measures the execution time. Table A.1 shows the subtasks as well as the measured times when executing the test with db4o-old. The numbers for db4o-old from Table A.1 are also plotted in Figure 4.1.

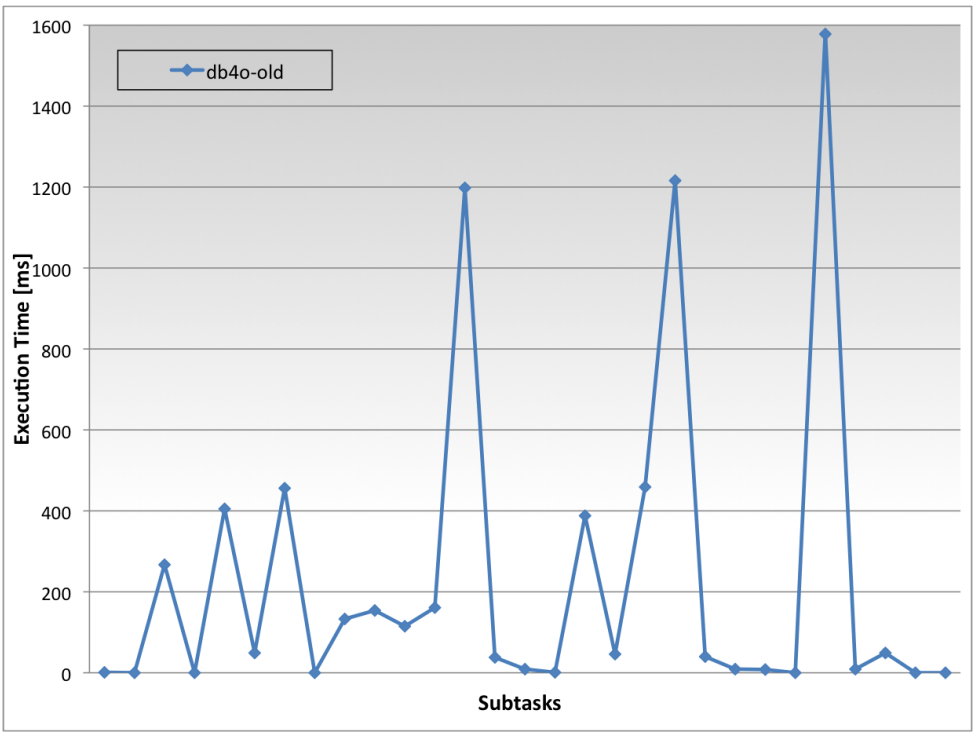

Figure 4.1: A plot of the numbers for db4o-old in Table A.1.

It is clearly visible that operations involving the creation of collections have the worst performance: Each of these three operations takes more than one second to execute. Also very long take the creation of object types and dressing an object with such a type (both around half a second). For all these operations several checks are carried out by Avon to ensure the existence of all objects involved and that they all have the correct types. These checks often are conducted by accessing information from the storage layer, which then translates to database accesses.
On the other hand, adding members to extents takes very little time because it is a pure Java-operation not involving any queries to the database.

4.2 Representing Data at the Storage Layer

The concepts of OM, as implemented in Avon's model layer, have to be mapped to another model that is better suited to address the requirements of the storage layer. The meta-model

derived for that purpose will be discussed in Section 4.2.1. Then, Section 4.2.2 shows the first translation of that meta-model to a usual Java model. Finally, the problems of the Java model when used with db4o will be highlighted and an improved model for the db4o-setting will be demonstrated in Section 4.2.3.

4.2.1 The Meta-Model

As we know from OM, objects can be dressed with types. The reverse operation is stripping a type from an object. Such a type instance is represented by an `InformationUnit`, which is, in fact, just an array of attribute values. Its meaning is described by a `TypeDescriptor` specifying the types of each attribute by position.

For illustration of the concepts of information unit and type descriptor, we would like to model a type person that consists of a name and a birthdate. The name itself should consist of a first and a last name, both of type string. The birthdate should consist of three integers: A year, a month and a day.
Defining these types and creating an instance of our person type named "Max Payne" and born on March 21, 1972 leads to the structure depicted in Figure 4.2.
On the left we have a `PersonUnit`. Its descriptor, the `PersonDescriptor`, tells us, that the unit contains two attribute values: One described by a `NameDescriptor` at position `0` and one described by a `DateDescriptor` at position `1`. As we can see, the `PersonUnit` is in fact linked to a `NameUnit` and a `DateUnit` that in turn are each linked to the respective descriptor. These descriptors tell us, that a name is made up of two values of type `String` (that would be the first and the last name) and that a date consists of three values of type `int` (year, month and day). The units themselves then contain the values described by these descriptors.
On the right side of the figure, we see that for a type descriptor there is also an information unit. These units contain as first value the type's name and as second value its attribute types. The units are linked with the same identifier as the type descriptor (eg. `oid1`).

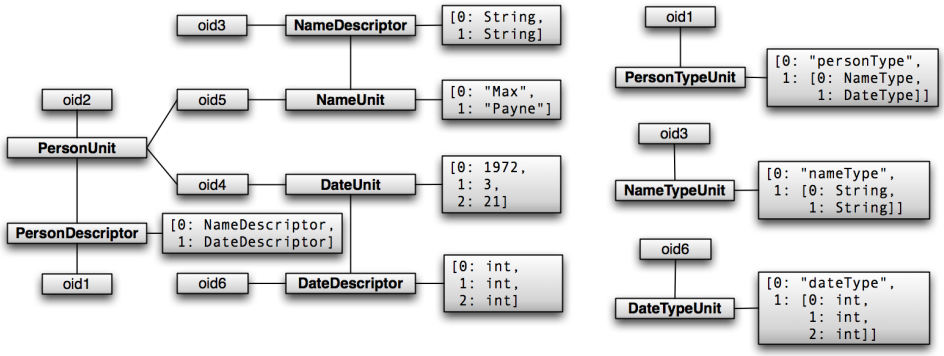

Figure 4.2: Example for the use of type descriptors and information units.

The same concept holds for an `Extent`. Extents are the representation of collections on the storage layer. An `Extent` is therefore just a collection of values. The semantics are again

given by a `TypeDescriptor`. As they have different properties than objects, extents are actually described by a sub-type of `TypeDescriptor` called `ExtentTypeDescriptor`, while object type instances are described by an `ObjectTypeDescriptor`.

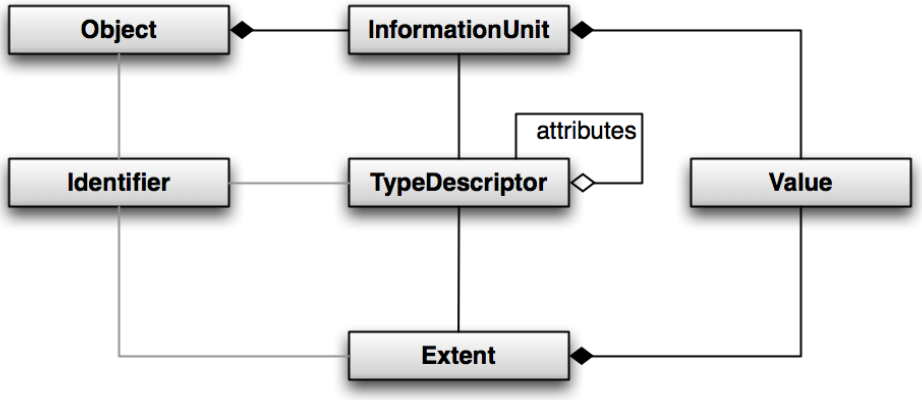

Figure 4.3: Data model at the storage layer.

Figure 4.3 shows the storage layer's meta-model with the relations between objects, extents, information units and type descriptors. The actual `Value` stored in an `Extent` or `InformationUnit` can be one of the following:

- Identifier pointing to an `Object`.
- Identifier pointing to an `Extent`.
- Basic value (String, int, boolean, ...).

Instances of `Object`, `TypeDescriptor` and `Extent` are identified by an `Identifier`. Most of the time only these `Identifier` instances are given from the storage layer up to the model layer while the other concepts stay encapsulated in the storage layer.

4.2.2 Representing Data – The Java-Approach

When implementing the meta-model in Figure 4.3 in Java, the result for db4o-old looked like Figure 4.4. From the viewpoint of a Java developer with knowledge of object-oriented software design, this is in fact a nice and straight-forward way of implementing the meta-model.
The question now is, whether the implementation is well-suited when used in combination with db4o. As mentioned in Section 3.3, db4o is well capable of storing any object graph created in Java. The problems – or better, inefficiencies – arise when starting to write queries accessing the data stored in db4o.

Assume the query shown in Listing 4.1. The navigation path in Java, and hence the access path for SODA queries in db4o, will be from `StorageObject` to `InformationUnit` to `InformationUnitElement`.

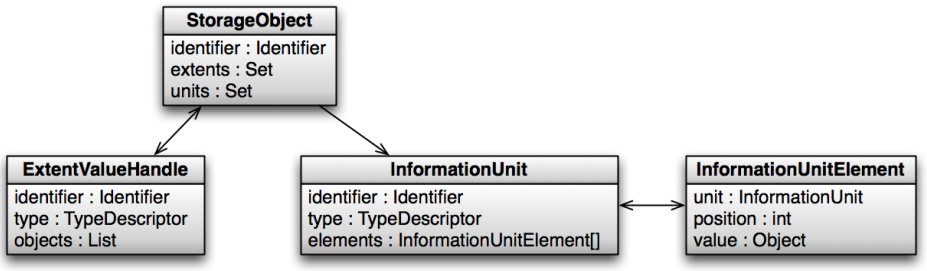

Figure 4.4: Implementation of the data model in db4o-old.

```
Give me all objects that are of type T
and have at attribute position P
the value X.
```

Listing 4.1: Simple query for objects by type and attribute value.

The fields through which the query will descend are `StorageObject.units` and `InformationUnit.elements`, of which the former is a `Set` and the latter an `Array`. While db4o supports queries on collections and arrays, there are three issues encountered while analysing db4o-old.

First, queries on collections and arrays are slow, just due to the way they are processed by db4o. The second issue is writing queries for collections and arrays. It is possible to write SODA for those field types, but it can be very hard – or not very intuitive. The people implementing db4o-old were not able to find working SODA expressions for all queries that had to be implemented. For the left over cases they resorted to using Native Queries.

Native Queries are not per se something bad that should be avoided by all means. As we have seen in Section 3.3.1, they provide more programming language integration and safety than SODA as all of the field names used in the query are written as Java expressions and therefore statically checked at compile time. But we also mentioned that db4o's internal translation from Native Queries to SODA can turn out unoptimised in which case query execution will take longer.
Therefore, only if we are able to write our query in SODA, can we be sure that it will *always* be executed in the fastest way possible.

```
class Person {
   String name;
   Set<String> phoneNumbers;
}
```

Listing 4.2: Class `Person` containing a name and a collection of phone numbers.

The third issue mentioned above is about indexing. In db4o, indices are created on fields of classes. If we for instance have a class `Person` as shown in Listing 4.2, we would tell db4o to index the name field of that class by adding the code shown in Listing 4.3 to our db4o configuration.

```
Db4o.configure().objectClass(Person.class)
    .objectField("name").indexed(true);
```

Listing 4.3: Configuring db4o to index the `name` field of class `Person`.

There is one limitation though: db4o cannot index the elements of a collection or an array. Therefore trying to index the `phoneNumber` field of our `Person` class as shown in Listing 4.4, will not bring any benefits – however, db4o will not complain if we try to do so.

```
Db4o.configure().objectClass(Person.class)
    .objectField("phoneNumbers").indexed(true);
```

Listing 4.4: Configuring db4o to index the `phoneNumbers` field of class `Person`.

What we could do instead, is create our own class of entry for the phone number collection (Listing 4.5).

```
class PhoneNumber {
    String number;
}
```

Listing 4.5: A wrapper class for a phone number.

Changing the type of `Person.phoneNumbers` from `Set<String>` to `Set<PhoneNumber>` will then allow us to index the field `PhoneNumber.number` (Listing 4.6). Thereby, queries on phone numbers stored in the collection of a `Person` instance will profit to some degree from that index. However, this index will contain *all* instances of `PhoneNumber`, and not just those in the collection of the current `Person` object.

```
class Person {
    String name;
    Set<PhoneNumber> phoneNumbers;
}
```

Listing 4.6: Class `Person` containing a name and a collection of phone numbers – now represented by `PhoneNumber` instances.

Having identified the three issues of slow and unintuitive queries as well as lacking indexing on collections and arrays, we are now prepared to derive an optimised implementation of the meta-model that's better suited for the db4o-environment.

4.2.3 Representing Data – The db4o-Approach

As discussed in Section 4.2.2, the main performance problems in db4o-old seemed to be arising from issues around collections and arrays. The obvious countermeasure is to get rid of all collections and arrays. Luckily, there is a way to do so, which consists of substituting a collection of children in the parent object with a parent-reference in the child. The parent object then no longer knows all its children, but every child knows its parent. Applying this approach to the references between `StorageObject`, `InformationUnit` and `InformationUnitElement` – leaving the association between `StorageObject` and `ExtentValueHandle` as is for the moment – leads us to the new implementation shown in Figure 4.5.

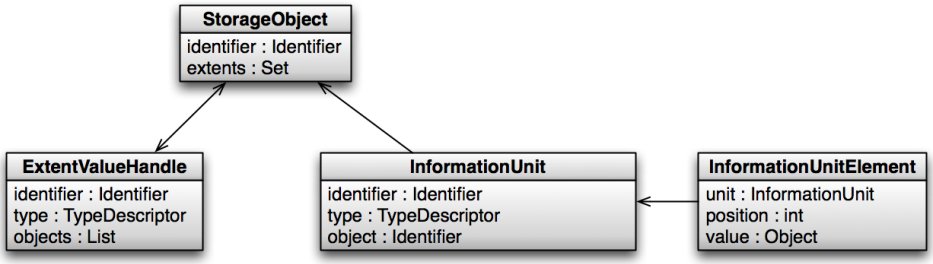

Figure 4.5: Redesigned implementation of the data model in db4o-awesome.

As one can see from that Figure, the access path has been reverted in comparison to Figure 4.4: We now navigate from `InformationUnitElement` up to `StorageObject`.
But since we no longer have any arrays or collections along that path, it is straight forward to write SODA expressions for all queries.

Having implemented the improved design from Figure 4.5 and having rewritten all queries to SODA brings us to *db4o-awesome 1.0*. The resulting performance achieved when running `OMPerformanceTest` with this implmementation is shown in Table A.1 and Figure 4.6.
The results show that getting rid of all collections and arrays and rewriting all queries to SODA resulted in a total execution time that is only about 50% of what we had in db4o-old. These improvements were achieved at every subtask of the test, which indicates that every operation on the model layer seems to profit from the changes we made at the storage layer.

4.3 Redesigning Extents

As mentioned in Section 4.2.3, we spared the association between `StorageObject` and `ExtentValueHandle` from changes for the time being. Let us now see, if we can apply the lessons learned during the previous sections to that association.
The association in question is actually an M:N relationship: Each `StorageObject` can be part of zero to N extents, while each `ExtentValueHandle` can contain zero to M objects. That is why we have collections in both of the classes.

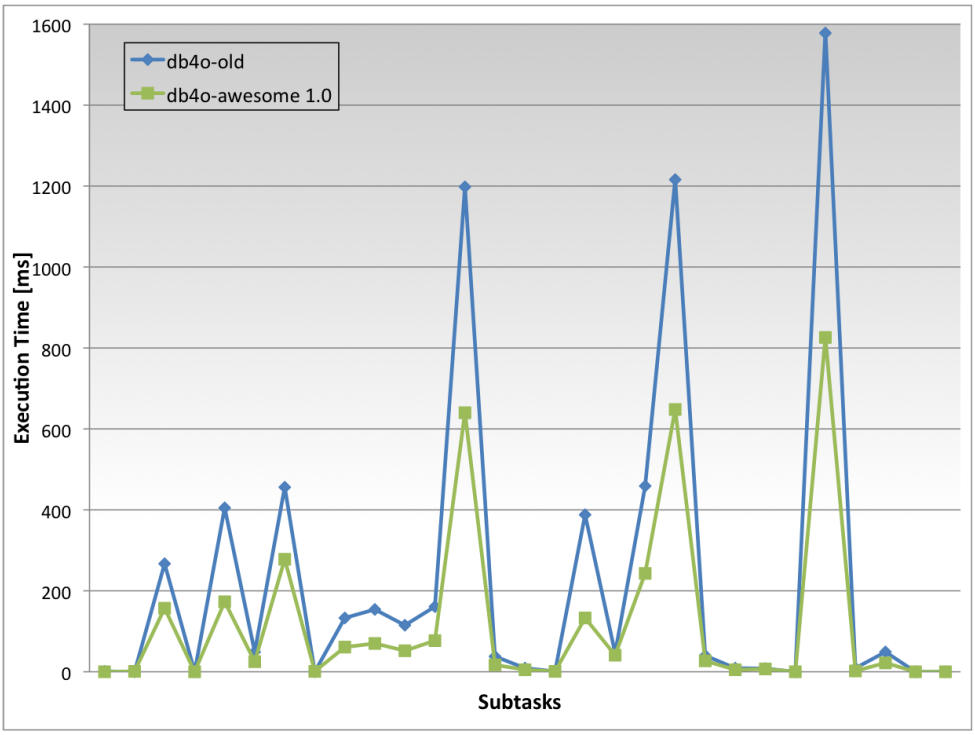

Figure 4.6: A plot of the numbers for db4o-old and awesome 1.0 in Table A.1.

Sticking to our mantra of getting rid of collections and arrays to improve performance, we would come up with a new design as shown in Figure 4.7. Clearly, this is exactly how one would represent such an M:N relationship in a relational database.

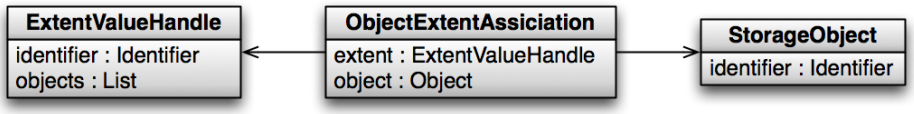

Figure 4.7: Redesign of the association between `StorageOject` and `ExtentValueHandle` to avoid the use of collections.

To see how the new design stacked up against the old version with the collections, a test suite was implemented. It would first create one thousand objects and a hundred extents and then randomly put objects into extents and also put some extents into other extents.
After that, it would run the queries of `contains`, `containsAll` and `size` on the extents, as well as add objects to and remove objects from extents and clear extents from all content.

The results gained from running this test suite revealed the following points:

- *Insertion of objects into extents is faster using the new design.*
  This is due to the fact, that – using the old design – for every insertion of one object into an extent during the tests, the whole extent with its collection had to be retrieved

from db4o, then the object was added to the collection and in the end the whole extent was written back to the database.
In contrary to that, inserting one object into an extent using the new design translates to creating a new instance of `ObjectExtentAssociation` and storing it in the database – which is clearly the smaller effort.

- *Queries like `contains` and `size` take less time to execute on the old design.*
  Here the collection pays off, because it's the only object that needs to be retrieved from the database. Once we have the collection, it will answer our queries, because it already supports these operations.
  Using the new design however means, that we have to inspect several instances of `ObjectExtentAssociation` looking for the object(s) in question.
  It is also possible, that once the collection was retrieved during one query, it still is available in db4o's cache when the next query is evaluated on the same collection, as the database was not closed after each query.

- *Configuring the right field indices in db4o results in big performance gains.*
  Setting up indexing, especially for the new collection-less design, has brought a speed-up of factor ten for the execution of the whole test suite.

Considering the first two points in combination with the aspect of OMS Avon being a *read-mostly* system, the conclusion of these tests with both designs was to stick with the old collection version. This way, the queries, who are dominating insertions, would run quicker.
The last point concerning indexing will be the next section's topic.

4.4 Getting Indexing to Work

As mentioned in Section 4.2.2, collections and arrays cannot be indexed with db4o. After getting rid of those two constructs with the redesign in Section 4.2.3, indexing should now be possible on all the fields used in the SODA access path.

```
public Query sampleQuery(TypeDescriptor type, int position,
    Object value) {
    Query query = this.db.query();
    query.constrain(InformationUnitElement.class);
    query.descend("position").constrain(position);
    query.descend("value").constrain(value);
    query.descend("unit").descend("type").constrain(type);
    Query objectQuery =
        query.descend("unit").descend("object");
    return objectQuery;
}
```

Listing 4.7: SODA query to retrieve all objects of given type that have at given attribute position a given value.

Let us go back to the query in Listing 4.1 that we already discussed during Section 4.2.2. Expressing this query in SODA will result in what is shown in Listing 4.7.
This shows quite nicely which fields we have to index to accelerate the query – namely all the fields that are part of `descend` calls. In our example, these fields would be `position`, `value` and `unit` of class `InformationUnitElement` and the fields `type` and `object` of class `InformationUnit`.

However, indexing all the fields used in all the SODA queries did not result in the expected performance increase at first. Investigations using db4o's diagnostics possibilities showed, that most of the indices were in fact never even created.

4.4.1 Indexing and Interfaces

As mentioned at the end of Section 4.2.1, the communication between storage and model layer is largely based on identifiers. In fact, there is a whole type hierarchy of identifiers defined by several interfaces in Avon. As it is common and good practice, every field or variable that was supposed to hold an instance of such an identifier was declared to be of the interface type. This principle, known as *programming against interfaces*, is shown in Listing 4.8: Event though we know in class `StorageObject`, that the instance stored in the field `identifier` will be of type `IdentifierImpl`, we declare it to be of the interface type `Identifier`.

```
interface Identifier {
   int getId();
}

class IdentifierImpl implements Identifier {
   private int id;

   int getId() {
      return this.id;
   }
}

class StorageObject {
   Identifier identifier;  // not IdentifierImpl !
}
```

Listing 4.8: An example for programming against interfaces.

Suppose we want to query for instances of `StorageObject` by a given `Identifier`. Listing 4.9 shows a query that will produce these results. For shortening this query's execution time, we would like to index the `identifier` field of class `StorageObject`. In order to create this index, we add the line shown in Listing 4.10 to our db4o configuration code.

```
public ObjectSet storageObjectById(Identifier identifier) {
    Query query = db.query();
    query.constrain(StorageObject.class);
    query.descend("identifier").constrain(identifier);
    return query.execute();
}
```

Listing 4.9: Query for `StorageObject` instance by given `Identifier`.

```
config.objectClass(StorageObject.class)
    .objectField("identifier").indexed(true);
```

Listing 4.10: Indexing field `identifier` of class `StorageObject`.

Even though db4o does not complain when we add this index, it will in fact never be created. The problem is, that the field we are telling db4o to index, has the declared type of an interface. And as you can see from the definition of the interface `Identifier` in Listing 4.8, it contains absolutely nothing that could be indexed.
If we really want to index the `identifier` field of our class `StorageObject`, we have to change its declared type to `IdentifierImpl` (Listing 4.11). Only then will the index declared in Listing 4.10 be created.

```
class StorageObject {
    IdentifierImpl identifier; // not Identifier anymore !
}
```

Listing 4.11: Changing field type from interface to implementation.

We then also have to change the signature of `storageObjectById` in Listing 4.9 to accept a parameter of type `IdentifierImpl` instead of `Identifier`. After completing this refactoring, our query can finally benefit from the indexed field in `StorageObject`.

4.5 db4o-awesome 2.0

Turning on all the useful indices and applying the findings described in the previous section lead to version 2.0 of the db4o-awesome implementation.
As you can see from Table A.1 and Figure 4.8, the correct configuration of indexing really paid off. The execution time of each of `OMPerformanceTest`'s steps has again been cut by 50% compared to db4o-awesome 1.0. This results in an overall execution time of the whole test case of just below 1.3 seconds as compared to 6.7 seconds achieved with db4o-old. While this is still ten times larger than the result from the in-memory implementation, it is only a fifth of the time posted by db4o-old prior to all the optimisations described in this chapter. In other words: The redesign described in Section 4.2.3 and the changes implemented to enable indexing where necessary as described in Section 4.4.1 resulted in a total performance gain of 500%.

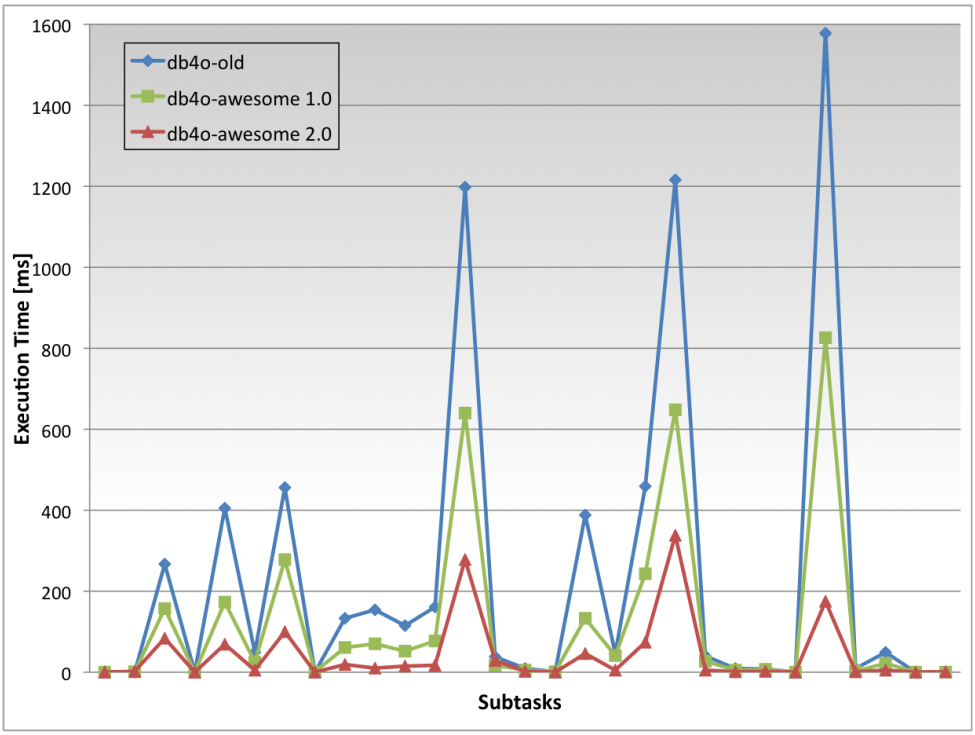

Figure 4.8: A plot of the numbers in Table A.1.

It is noteworthy that all the numbers presented in Table A.1 were obtained with db4o version 7.4.71. Running the same test again later, after switching to db4o version 7.8.82 to profit from bugfixes, gives a different picture. While the ratios between the three versions old, awesome 1.0 and 2.0 are about the same, the absolute values have gone up considerably, as Table 4.1 shows.

	Execution Time [ms]		
db4o Version	**old**	**awesome 1.0**	**awesome 2.0**
7.4.71	6789	3512	1283
7.8.82	~9000	~4600	~2700

Table 4.1: Execution times measured by `OMPerformanceTest` using different db4o versions.

4.6 Conclusions

Version 2.0 of db4o-awesome marks the final stage of optimisations in this work. Further optimisations with options provided by db4o would possible. For instance, a `CachingIoAdapter` could bring additional performance gains by caching I/O operations. Or, index access time could be decreased with the right parameters for B-Tree height and node size.

However, it is not the idea of Avon's architecture to perform caching at the storage layer. Such caches should be implemented on the upper layers. Thus, Avon will profit from the caches, no matter what implementation of the storage layer is used.
The options mentioned for B-Tree tuning, as well as other options available in db4o, only increase performance if the values used are very well tailored to the system (processor, chipset, memory) the application is executed on. Optimising these settings for one system could therefore result in performance degradation on other hardware.

While discussing the performance of db4o-old in Section 4.1 we highlighted that operations involving the creation of collections had the highest execution times. With db4o-awesome 2.0 this is still the case. In fact, the proportions between the quicker and the slower operations have not changed. It merely looks as if we had just run the test on faster hardware, which lead to a decrease of execution time for all operations.
We mentioned the presence of several checking and assertion mechanisms in Avon as possible cause for the slower operations. The vast part of these mechanisms is still in place in db4o-awesome. Their execution has just been sped up by the more efficient underlying implementation. This means that inspecting all the checking and asserting could identify additional optimisation potential to increase the performance of slow operations – like collection creation – even further.

5 Indexing in OMS Avon

Section 4.4 discussed enabling indexing in db4o and the performance gains associated with this step. Even though we have now done everything possible on this level, there still is room for further improvements through indexing. However, this additional indexing cannot be done by db4o itself. This is due to the fact that our data model at the storage layer is too general to further leverage db4o's indexing capabilites.

At the beginning of this chapter some queries that would benefit from additional indices will be discussed in Section 5.1. Section 5.2 will show the implementation of an index structure to address one of the queries identified in Section 5.1. Section 5.3 then discusses the index structure's incorporation into OMS Avon. Finally, Section 5.4 presents the results achieved with this new index structure.

5.1 Example Queries

In this section, we have a look at two types of queries that could be executed faster with the support of additional indices.

5.1.1 Selection Queries by Attribute Values

We have already seen a case of a selection query by attribute value in Listing 4.1. Actually, as in Avon all objects are always part of some collection, in reality there is one additional constraint concerning an `Extent` on this kind of queries. Therefore the real world example of such an attribute query would be what is shown in Listing 5.1.
In order to answer such a query, Avon would retrieve the extent, iterate over its members and check for each, if it matched the query. With the array- and collection-less data model presented in Section 4.2.3, our access path for this matching starts at the attribute value and from there goes up to the object. Assuming that on average there are more attribute values

```
Give me all objects in Extent E
that are of type T
and have at attribute position P
the value X.
```

Listing 5.1: Full query for objects by extent, type and attribute value.

than objects in an Avon database, it would however be more discriminating on the size of the result candidate set, if we could start at the object and descend to the attribute.
The index on attribute values configured in db4o certainly helps a lot here to keep access time at a reasonable level. But this index contains *all* attribute values of *all* objects in the database. Presumably, this means that this index will very grow large.

Examining the query in Listing 5.1 we see that in fact we are only interested in the attribute values of objects in a given extent and of a given type. Therefore the optimal index for such a query would contain all attribute values of attributes at given position, belonging to objects of given type that are in given extent.
Such indices on attribute values are usually implemented with B+-Trees in a DBMS. The implementation of such an index will be the topic of the upcoming Section 5.2.

5.1.2 Queries for Collection Membership

A second example for queries that could benefit from additional indexing in Avon are queries for collection membership. Such a query is for instance the one shown in Listing 5.2. The exisiting implementation for such queries in Avon is quite simple – it compares all entries of the first extent with all entries in the second to find matches (also known as *nested loop join*).

```
Give me all objects of type T
that are member of extent E1
and are member of extent E2.
```

Listing 5.2: Intersection query for members of two extents.

As with queries by attribute values presented in the previous section, queries for collection membership are nothing special in the world of DBMS. Therefore there is already an index structure known to accelerate such queries for collection membership: The bitmap index.
The implementation of such indices is planned for the near future of Avon's development.

5.2 Implementing a B+-Tree

A B-Tree is a data structure known since the early seventies of the twentieth century. B-Trees are taught in the second semester of computer science studies at ETH during the lecture

”Algorithms and Data Structures”. The ”plus” version as special case of B-Trees is mentioned there as well.

B^+-Trees are used in DBMS because their fix node size can be set to a value that corresponds to the block size on disk – where the index will be saved persistently. Often, the tree’s leaves are linked with their neighbours in both directions to facilitate range queries.
Being a search tree, a B^+-Trees basic operations are insert, remove and search.

5.2.1 Search

Implementing search is straight forward: Navigating through the inner nodes by comparing keys and descending towards the leaf containing the desired entry.

5.2.2 Insert

Insertion requires a little more work. Because of the maximum node size defined by the tree’s order, a node may be too big after insertion. In this case, the node is split and half of the keys and children are moved to a new node. To separate these two nodes, an additional separator key must be inserted in the parent node. This can in turn cause the parent to be too big and therefore propagate splitting upwards in the tree. The tree grows in height if this splitting process reaches the root and a new root is created.

5.2.3 Delete

Deletion is the trickiest of the three operations to get right when implementing. A good recipe – and also a nice overview for B^+-Trees in general – is given by Jannink in [1]. The essence of his algorithm is quite simple. If a node becomes too small after deleting from it, there are two operations to solve the problem:

- *Redistributing among neighbours.*
  If either the node’s left or right neighbour contains more keys than the margin of subsistence, some of this neighbour’s keys (and children if it is not a leaf) can be shifted to the current node. Then the key in the ancestor node separating the path between the two nodes has to be updated to reflect the shift.

- *Merging with a neighbour.*
  If both neighbours contain only the minimum number of keys necessary, the node has to be merged with one of the neighbours. After the merge it is also necessary to remove a key from or replace a key in the ancestor node. If it has to be removed, this ancestor node can in turn be too small and therefore deletion must be propagated upwards in the tree. If this process reaches the root, it will be collapsed and replaced with its only child left. This means that the tree shrinks in height.

While descending recursively from the root to the leaf where deletion should take place, the algorithm also keeps track of each node’s neighbours in the tree. This information can then be used if an underflow occurs after deleting from the leaf to re-balance the tree while unwinding the recursion.

The implementation's most delicate bits were selecting the correct replacement for the separating key in the ancestor and shifting the right amount of keys and children to the neighbour in all cases (merge means shifting all entries).

According to [1], several B+-Tree implementations in DBMS avoid really removing nodes from the tree due to deletion by performing so-called *lazy deletion*. This approach never, or only after a certain time of emptiness, removes nodes from the tree. This is preferable when optimising the tree for being stored on and retrieved from a hard disk. Removing a node would introduce the overhead of having to rewrite the structure on disk. If a new node at the same position in the tree gets allocated again later, the structure must be rewritten again.
To circumvent this constantly changing structure on disk, the implementations using lazy deletion accept a certain degradation of the tree structure to avoid rewriting the index structure on disk.
As we do not have any control over how our tree will be stored in db4o, the idea of lazy deletion was not considered useful for our implementation.

5.2.4 Keys and Values

The simplest collection (and therefore extent) in OMS Avon is the set: It has no ordering and does not allow duplicate entries. However, we are not using our B+-Tree for indexing *objects* in an extent, but merely the object's *attributes*. This means that even though the set does not have to handle duplicates, our index still needs to do it. Suppose we have a person type that defines a person as having a name and an age. Therefore the two instances of such a type shown in Listing 5.3 would be perfectly valid. These two instances are truly distinct and hence adhere to the properties of a set. But at the same time, they both contain the same value in the `name` attribute.

```
person1 = {name="Christoph", age="24"}
person2 = {name="Christoph", age="26"}
```

Listing 5.3: Two instances of a person type containing the same attribute value.

To cope with such duplicate attribute values, our index uses the value itself as key for the entry. Together with that key it stores a list of object identifiers containing the identifiers of all objects having the same attribute value.

5.2.5 Testing the Implementation

As it has to be done for all code in OMS Avon, unit tests needed to be written for the B+-Tree. This task is however not as straight forward, as it may sound. The problem is that one is lacking access to the tree's inner workings. After all, such a search tree is an *abstract data type (ADT)* who's actual implementation details *should* be hidden. Two possible solutions for this situation would be:

- Using a special Java class loader one could change the visibility of the private methods inside the tree and make them public. A tree instance loaded with this class loader could then be tested in the usual way.

- Teaching the ADT the ability to test itself. This is quite simple for a B^+-Tree, because the data structure is built from a small set of constraints. Running a large amount of randomised cycles of insert, remove, and search operations with self tests in between can then show occurring inconsistencies.

The second approach bears the same uncertainty as testing generally bears: If the randomised tests don't show any problems this does not mean, that there aren't any.
However, it is easier to ensure the properties of a B^+-Tree than to figure out all the special cases for each internal operation and constructing a test case for all of them. Therefore the self-testing approach was chosen and implemented.

Some of the checks implemented to tell whether a tree node is healthy are:

- Checking the order of the keys in the node. They should be ordered ascending.
- Comparing the key- and the child-count. Each key should separate two children. This means that a non-leaf node containing k keys should have $k + 1$ children.
- Comparing the keys in the children to the keys in the parent. All keys in the child should be smaller or equal the right key in the parent and greater than the left key in the parent.
- Checking the `next` and `previous` references on the leaf level.

These checks turned out to be a handy tool when tinkering with all the boundary conditions of deletion.

5.3 Incorporating Indexing into Avon

Having implemented an index structure as outlined during Section 5.2, this structure can now be incorporated into Avon to – hopefully – benefit from lower execution times for queries. The integration consists of the following three subtasks:

- Providing the functionality to create and remove indices on the OM layer.
- Maintaining existing indices by subscribing to storage events of the operations dress, strip, setAttributeValue and setAttributeValues on objects as well as insertion into and removal from extents.
- Using index accesses instead of the usual selection implementation during query execution whenever possible.

The index structures themselves are held by the storage layer which is also responsible for persisting and reloading them.

5.3.1 Managing Indices on the OM Layer

On the OM layer, indices can be created, retrieved and removed. All these operations are shown in Listing 5.4.

```
// Creating an index.
db.storage().createIndex(txHandle, indexId, extentId,
   objectTypeId, attributePosition);

// Retrieving the index.
Index index = db.storage().getIndex(txHandle, indexId);

// Second method for retrieving the index.
Index sameIndex = db.storage().getIndex(txHandle,
   extentId, objectTypeId, attributePosition);

// Removing the index.
db.storage().removeIndex(txHandle, indexId);
```

Listing 5.4: Creating, retrieving and removing indices on the OM layer.

For creating an index, we need the triple of extent, object type and attribute position to know what should be indexed. Additionally, an identifier for the index itself must be provided. With this index identifier we can later retrieve the index or remove it from the database. For retrieving there is also the possibility to provide the same triple as used for creation instead of the index identifier.

Retrieved indices should only be used to check, whether an index exists or perhaps to get its current size. Even though the index interface would allow to insert and remove entries as well as clear the whole index, this should not be done on the OM layer. The maintenance of indices will be handled automatically on a lower layer, as we will see next.

5.3.2 Index Maintenance with Storage Events

During his master thesis, Christoph Lins implemented an event module for OMS Avon [2]. This module allows us to get notified of different events in the system and to react to them if necessary.

When opening a storage, we register an event listener with the event system. This listener will be notified of all events occurring in Avon. As mentioned in the beginning of Section 5.3, we are interested in the events of the following operations taking place to keep our indices up to date:

- *Dressing an object with a type.*
  When dressing, values for all the attributes defined by the type have to be provided. Therefore, we have to add the new values to all existing indices on extents that the object is part of.
- *Stripping an object from a type.*
  This operation is inverse to the dress-operation. Hence the attribute values belonging to this type instance have to be removed from the indices they are part of.

- *Setting attribute value(s).*
  Setting an attribute to a new value means that we have to remove the old value from all indices and instead insert the new value.

- *Adding an object to an extent.*
  When an object is added to an extent, we have to check for all attributes defined by all types this object is dressed with, whether there exists an index. If so, we insert the new value into the index.

- *Removing an object from an extent.*
  Removal from an extent requires the same check for existence of indices as when adding to an extent. If an index exists, we remove the entry for this object.

Additionally there will be an event of creating a new index. The reaction to this event is to add already existing members of the extent that have the correct type to the new index.
Our event listener will only react to the events we have just discussed. All other events that are not relevant for index maintenance are ignored. The listener is therefore called `IndexRelevantStorageEventListener`.

5.3.3 The `ComparisonIndex` Interface

The selection queries intended to be replaced with index accesses are modeled with the help of the `ComparisonPredicate` class. Such a predicate contains a `ComparisonOperator` and a value to be compared to. The `ComparisonOperator` can be one of the following operators: *equal*, *greater*, *greater or equal*, *less*, *less or equal*, *not equal*, *like*, *in*, *subset*, *superset*.
Given the right index instance, the predicate therefore contains all information needed to produce the result of the index access.

The interface derived for indices answering our selection queries reflects this observation. As shown in Listing 5.5, this `ComparisonIndex` interface defines only one method. Given a predicate it will return an iterator for the result set.

```
public interface ComparisonIndex extends Index {

   Iterator result(ComparisonPredicate predicate);

}
```

Listing 5.5: The `ComparisonIndex` interface.

Implementing this interface, our B$^+$-Tree checks the operator in the given predicate to forward the call to the appropriate method. However, the tree does not support all kinds of operators: *in*, *subset* and *superset* deal with collections and are not suited for the single values stored in a tree. The result for all other operations is produced by traversing the tree's leaf level with either an entry, exit or filter condition on the keys stored there.

5.3.4 Modifying the Query Tree

As pointed out in the the overview at the beginning of Section 5.3 we would like to use existing indices whenever possible to answer selection queries. For ordinary query execution, first a tree of the query's operations will be built. An example for such a query tree is shown on the left in Figure 5.1: This query will return the union of friends at age 26 and movie stars named Christoph.

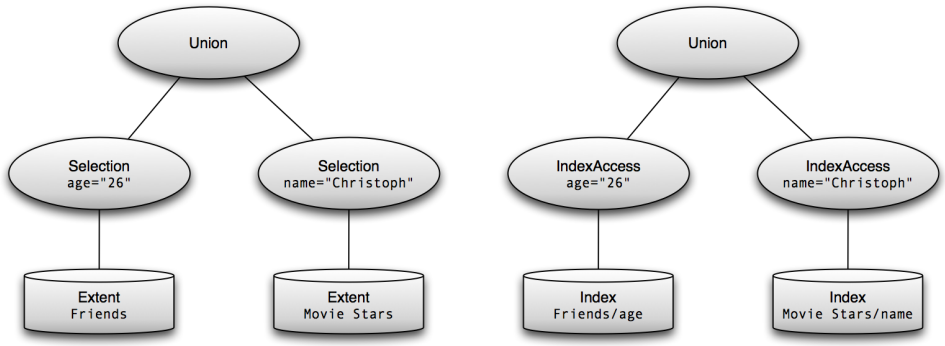

Figure 5.1: Example query tree and modification by the `OptimiserVisitor`.

Such a tree will be visited by an `EvaluatorVisitor` to produce the query's result. The visiting process starts at the tree's root. The result for a subtree is calculated by first visiting the root's children and producing their results before using these results to produce the result of the whole subtree at its root. In our example this means that the visitor first evaluates the two selections on their respective extents and then uses these two results to compute the union at the query tree's root.

In order to replace selections where possible, a new `OptimiserVisitor` was implemented. This visitor also visits all nodes in the tree. But instead of calculating a result, it looks for `Selection` nodes. If such a node uses a `ComparisonPredicate` for an existing index, the visitor will create a new `IndexAccess` node with that predicate and put it at the place of the `Selection` node in the query tree.
Suppose that we have configured an index for the age attribute on our friends extent and an another index for the name attribute of the movie stars extent. The `OptimiserVisitor` would then modify our example tree to what is shown at the right in Figure 5.1: The two selection nodes are replaced by index access nodes that each make use of the appropriate index.

The `EvaluatorVisitor` was extended with the handling of `IndexAccess` nodes. When visiting such a node it will use the predicate to retrieve the result by calling the `ComparisonIndex`'s method discussed in Section 5.3.3.

5.4 Index Performance

In order to measure any changes in performance caused by the introduction of indices in Avon, we wrote an `IndexPerformanceTest`. Its scenario is to create one object type with one attribute and a collection for this object type. It will then insert several objects into the collection and measure the time taken for this insertion process – once without indexing and once with an index created on the collection for the object type's attribute.
The second part is to measure the time taken for running 20 queries on the inserted data – again, once without and once with an index. Fifty percent of those queries will be for non-existing attribute values and therefore return empty. The other half will produce a non-empty result set. Running the test with an object count of 100, 200 and 400 lead to the results shown in Table 5.1 and Figure 5.2.

	Execution Time [s]		
Object Count	**Insertion**	**Querying without Index**	**Querying with Index**
100	23.6	22.8	3.1
200	84.3	100.8	5.7
400	330.1	884.6	13.6

Table 5.1: Results of `IndexPerformanceTest`.

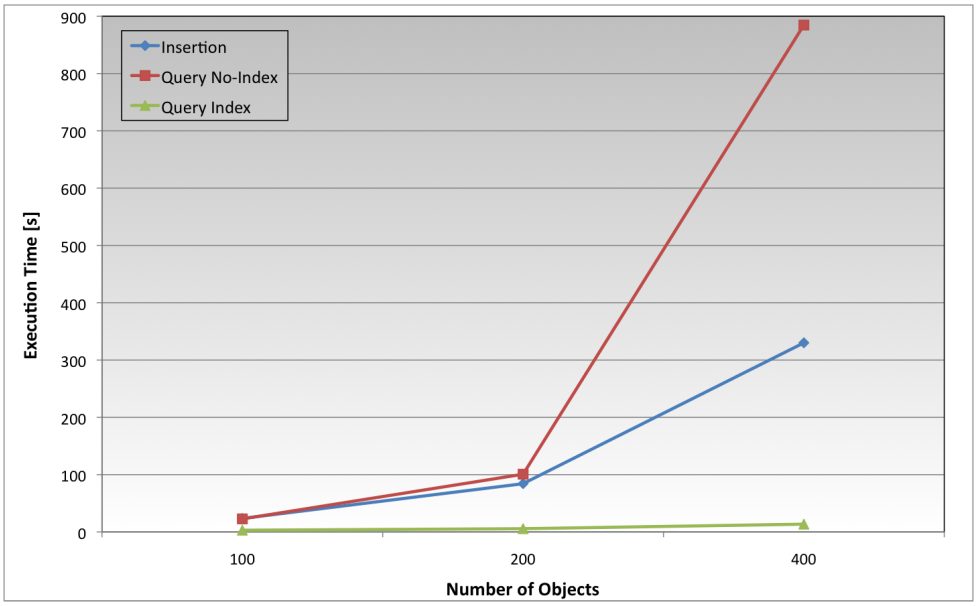

Figure 5.2: Results of `IndexPerformanceTest` (plot of Table 5.1).

The first thing to note is that in fact our work on the index paid off in query execution time. With 100 objects the time taken for 20 queries drops from around 23 seconds without index to about 3 seconds when using an index. At 400 objects the gap widens to 884 versus 13 seconds.

Insertion performance does not show any difference between having an index configured or not (hence only one graph for insertion in the figure). Even totally deactivating any index-related code in Avon – that would be the `IndexRelevantStorageEventListener` and the `OptimiserVisitor` – does result in the same insertion time. This does not mean that our implementation of index maintenance is extremely optimised. It is merely due to the fact that the rest of Avon performs so poorly that the additional overhead is just not visible.

The overall poor performance is clearly visible in Figure 5.2: All three curves plotted grow exponentially when linearly increasing the object count. This implies that there is some intrinsic flaw in Avon's implementation which remains to be investigated in the future.

6 Conclusions and Outlook

The work for this thesis comprises two major parts. The first one is the optimisation of Avon's db4o-based storage implementation. Having completed the thesis' original topic gave way to the second part of enriching Avon's query processing with the ability of using index accesses to boost performance.

Redesigning the db4o-based storage implementation resulted in considerable performance improvements. This was accomplished by identifying weaknesses in db4o's handling of collections and arrays and introducing a new design that works around these problems.

Indexing based on B^+-Trees and bitmap indices is an entry that already existed on the list of wanted features in Avon. There have been attempts of completing this task before, but none of them proved successful. The implementation of B^+-Trees used for the acceleration of selection queries by attribute value provided in this thesis can therefore be regarded as a successful tracer bullet for the whole task: It shows that the task is feasible and prepared the system for the full implementation.

Working on both parts provided insight into and understanding of many core concepts of the complex system that OMS Avon is. During the work, several ideas on how to further improve some of Avon's parts came up. These ideas are the topics of the following sections.

6.1 Unifying Checking

In the beginning of Chapter 4 we set forth the numerous checking mechanisms in Avon as one possible factor for performance issues. As we noted at the end of the same chapter, these checks have remained unchanged to a great extent and that they probably were bearing additional optimisation potential.

On the other hand, when analysing index performance in Section 5.4, we noted the unexpected exponential growth in execution time for a linearly growing object count. It is our educated guess, that the checks also played a crucial role in this behaviour.

At the moment, checks are executed at different levels in Avon. Moreover, different storage implementations perform different checks and throw exceptions on different occasions or with different messages. It is therefore favorable to factor out all checks into a dedicated checking package, positioned above the different storage implementations in Avon's architecture. This would ensure that, no matter what storage implementation is used, the system will always execute the same checks and throw the same exceptions.

Once factored out, the checking code could then be made configurable. One can imagine different levels of checking being enabled through this configuration. A simple switch for selecting *all checks on* or *all checks off* will then provide insight on the impact on performance all the checks have.

Should one in fact find that the checks spoil the system's performance to a considerable amount, it would be a viable option to speed-up the checks by caching information above the storage layer. Again, by having caches only above the storage, they will be available and work the same no matter what storage implementation is used.

6.2 Separating Typing and Classification in Indexing

In Section 5.3.2 we presented a list of events that are of interest for index maintenance. For the event of dressing an object with a type, we noted that all existing indices on extents that the object is part of possibly have to be updated. Compared to how indexing works in db4o, this is a considerable additional amount of work since db4o only has indices per type (class) and attribute (no additional extent-dimension).

The need for this additional amount of work stems from the fact, that OM separates typing and classification, while Java and therefore db4o do not. In the current implementation, this separation has not been applied to the index structures. If we had in addition to the B^+-Trees also bitmap indices available, we would be able to perform the separation also in indexing and therefore remove a part of the work for updating indices. This would be accomplished by having B^+-Tree indices per type and attribute just like in db4o. The additional extent dimension will then be handled with bitmap indices on the extents.

In this setting, a dress operation means that we only have to update the B^+-Trees for the new type's attributes – we do no longer have to care for extents. However, evaluating a selection query by attribute value on an extent (Listing 5.1) can then no longer be done by just one index lookup. The result must now be calculated by evaluating the `ComparisonPredicate` on the B^+-Tree and then intersecting the resulting list of object identifiers with the bitmap index on the target extent.

As it is already planned to implement bitmap indices in Avon in the near future, the basis for this proposed separation of typing and classification for indexing would already be provided. Changing the use of B$^+$-Trees to the new setting is a short task that mainly requires removing code that is no longer needed.

6.3 Continuous Performance Testing

In Section 4.5 we observed that changing the version of the underlying persistence provider of the storage layer can have a noticeable impact on the overall performance of Avon: By moving from db4o version 7.4.71 to 7.8.82 we experienced a noticeable performance drop.

In order to automatically detect such changes or to prevent gradual performance degradation during further development of the system, it would make sense to have performance tests as part of the continuous build system. Within these tests, one would have to define limits for a performance minimum. If the system turns out to be slower than this minimum, a warning should be issued to the developers.

6.4 A Sidenote on Implementing B$^+$-Tree Deletion

As it was mentioned in Section 5.2.3, deletion is a B$^+$-Tree's most challenging operation implementation-wise. Trying some examples by hand revealed several cases of problems that can occur. We were however not sure to have identified all possible cases and their solutions. Consulting some text books, we tried to find a pseudo-code recipe for the implementation of deletion. One of the standard readings on the topic of data structures [3] has a chapter about B-Trees but mentions the existence of B$^+$-Trees just in one sentence. A publication about the fundamentals of database systems [4] offered a discussion of B$^+$-Trees, including pseudo-code for insert and search operations. With regards to deletion, however, even this book only states that it is a tricky operation and shows two graphical examples, but no pseudo-code. [1]

Locating information about deletion in B$^+$-Trees on the web also bears its problems: Querying a search engine for "B+Tree" actually brings up information about B-Trees, but not the desired "plus" version. The problem is that this query is interpreted by the search engine as looking for documents containing *'b' and 'tree'*. To circumvent this limitation regarding the "+"-character, people writing about B$^+$-Trees on the web resorted to using the term *'bplus tree'* or *'bplustree'* instead. Searching for these terms finally yielded the desired information.

As it turnes out – and as it was to be expected – other people before noticed the lack of pseudo-code for deletion in B$^+$-Trees in the standard text books. But it was not until 1995 that this was addressed in a publication. Jannink filled this gap with his paper, where he presented a flowchart as well as complete pseudo-code in procedural C style for the deletion from B$^+$-Trees [1]. Translating his pseudo-code to an implementation in Java took some time, but finally provided us with a working deletion algorithm.

[1] We later discovered that with [5] a text book containing pseudo-code for deletion exists.

A
Tables

Subtask	Execution Time [ms]		
	old	**awesome 1.0**	**awesome 2.0**
Retrieving type 'string'	1	0	0
Creating attribute of type 'string'	0	1	2
Creating set of 'string'	267	157	84
Creating attribute of type 'set of string'	0	0	0
Creating object type declaring two attributes	405	173	69
Creating object	49	25	5
Dressing object with object type	456	278	100
Creating 2 attribute values	0	1	0
Setting attribute of type 'string'	133	61	19
Setting attribute of type 'set of string'	154	70	10
Retrieving attribute of type 'string'	115	52	15
Retrieving attribute of type 'set of string'	161	77	17
Creating collection	1198	640	278
Retrieving collection extent	38	17	29
Adding member to collection extent	9	5	2
Removing member from collection extent	1	1	0
Creating another object type declaring 0 attributes	388	133	46
Creating another object	46	41	5
Dressing other object with other object type	459	243	74
Creating another collection	1216	648	338
Retrieving extent of other collection	40	27	5
Adding object to collection extent	9	5	1
Adding other object to other collection extent	8	7	2
Creating cardinality	0	0	0
Creating association (creating relation collection as side effect)	1578	826	175
Retrieving relation collection	9	2	1
Retrieving relation collection extent	49	22	5
Creating relation collection member	0	0	0
Adding member to relation collection extent	0	0	1
Total time elapsed	6789	3512	1283

Table A.1: Execution times measured by `OMPerformanceTest` using db4o-old, db4o-awesome 1.0 and db4o-awesome 2.0.

List of Figures

List of Tables

Acknowledgements

I thank my supervising assistant Alexandre de Spindler for supporting me, whenever I needed it, and for letting me do what I wanted otherwise. Dr. Michael Grossniklaus also earned my gratitude for in depth and behind the scenes information on and around OMS Avon. Thanks also to Carl Rosenberger and the whole db4o developer team for insight into db4o and the lessons in real life software engineering I learned during my internship with the team. Last but not least I thank Professor Norrie for mentoring my studies and providing a group of people and an environment that allowed this thesis being interesting and enjoyable to me.

Bibliography

[1] Jan Jannink. Implementing Deletion in B+-Trees. *SIGMOD Rec.*, 24(1):33–38, 1995.

[2] Christoph Lins. Event-based Information Sharing. Master thesis, ETH Zurich, 2009.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT press, 2001.

[4] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley Reading, Mass, 2000.

[5] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill Science/Engineering/Math, 2003.