Performance investigation into selected object persistence stores

by

Pieter van Zyl

Submitted in partial fulfilment of the requirements for the degree
Magister Scientia (Computer Science)
in the Faculty of Engineering, Built Environment and Information Technology
University of Pretoria

21 February 2010

Performance investigation into selected object persistence stores

by
Pieter van Zyl

Abstract

The current popular, distributed, n-tiered, object-oriented application architecture provokes many design debates. Designs of such applications are often divided into logical layer (or tiers) - usually user interface, business logic and domain object (or data) layer, each with their own design issues. In particular, the latter contains data that needs to be stored and retrieved from permanent storage. Decisions need to be made as to the most appropriate way of doing this – the choices are usually whether to use an object database, to communicate directly with a relational database, or to use object-relational mapping (ORM) tools to allow objects to be translated to and from their relational form. Most often, depending on the perceived profile of the application, software architects make these decisions using rules of thumb derived from particular experience or the design patterns literature. Although helpful, these rules are often highly context-dependent and are often misapplied. Research into the nature and magnitude of 'design forces' in this area has resulted in a series of benchmarks, intended to allow architects to understand more clearly the implications of design decisions concerning persistence. This study provides some results to help guide the architect's decisions.

The study investigated and focused on the *performance of object persistence* and compared ORM tools to object databases. ORM tools provide an extra layer between the business logic layer and the data layer. This study began with the hypothesis that this extra layer and mapping that happens at that point, slows down the performance of object persistence. The aim was to investigate the influence of this extra layer against the use of object databases that remove the need for this extra mapping layer. The study also investigated the impact of certain *optimisation techniques* on performance.

A benchmark was used to compare ORM tools to object databases. The benchmark provided criteria that were used to compare them with each other. The particular benchmark chosen for this study was OO7, widely used to comprehensively test object persistence performance. Part of the study was to investigate the OO7 benchmark in greater detail to get a clearer understanding of the OO7 benchmark code and inside workings thereof.

Included in this study was a comparison of the performance of an open source object database, db4o, against a proprietary object database, Versant. These representatives of object databases were compared against one another as well as against Hibernate, a popular *open source* representative of the ORM stable. It is important to note that these applications were initially used in their *default modes* (out of the box). Later some *optimisation techniques* were incorporated into the study, based on feedback obtained from the application developers.

There is a common perception that an extra layer as introduced by Hibernate negatively impacts on performance. This study showed that such a layer has minimal impact on the performance. With the use of caching and other optimisation techniques, Hibernate compared well against object databases. Versant, a proprietary object database, was faster than Hibernate and the db4o open source object database.

Keywords:
Persistence, Performance, Benchmark, Object Oriented Database Management Systems (ODBMS), Relational Database Management Systems (RDBMS) , Object Relation Mapping (O-R or ORM), Hibernate, Optimisation Techniques, Cache, Lazy Loading, Eager Loading

Supervisor: Prof. DG Kourie (Department of Computer Science, University of Pretoria)
Co-Supervisors: Dr. Andrew Boake (Espresso Research Group, Department of Computer Science, University of Pretoria)
Degree: Magister Scientia

Acknowledgments

I would like to thank my brother and mother who has been so patient with me. Kobus, thanks for all the coffee and understanding when I was too busy to make dinner. Mom, thanks for supporting me and understanding the missed visits. To my father who always believed I could achieve anything and trusted us to make our own decisions in life. I miss the long chats about love, life and stuff.

To my valued friends and research buddies Morkel, Johnny and Runette with whom I could always discuss research issues. They where also always ready to read my *very bad first drafts*.

I would like to thank my study leaders from the University of Pretoria, Prof Derrick Kourie and Dr Andrew Boake for taking this research on with me and helping me. They sometimes played good cop and bad cop, one asking the difficult questions and the other defending with me. I always enjoyed the free food, coffee and discussions at the Blue Crane. I would also like to thank Dr Louis Coetzee at the Meraka Institute for all the time and effort he has put into reading and helping me with my dissertation as well as for his quick feedback. I am grateful to the Meraka Institute for the time and support they have given to me.

To my girlfriend: at last I don't have an excuse to stay in front of my PC over weekends.

Lastly for Mannequin, Elizabethtown, Flashbacks of a Fool and Alex & Emma. The movies that inspired and made me smile in the last few years.

Contents

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

List of Figures

List of Tables

Acronyms

CORBA	Common Object Request Broker Architecture
EJB	Enterprise Java Beans
HQL	Hibernate Query Language
JDBC	Java Data Base Connectivity
JDK	Java Development Kit
JDO	Java Data Objects
JPA	Java Persistence API
ODB	Object Database
ODBMS	Object Oriented Database Management Systems
ODMG	Object Data Management Group
OO7	Object Operations 7
OODB	Object-Oriented Database
OQL	Object Query Language
ORDBMS	Object-Relational Database Management Systems
ORM	Object Relational Mapping
OSS	Open Source Software
POJOs	Plain Old Java Objects
RDBMS	Relational Database Management Systems
SODA	Simple Object Database Access

SQL	Structured Query Language
TPC	Transaction Processing Performance Counsel
VM	Virtual Machine
VQL	Versant Query Language

Chapter 1

Introduction

"Too often I've seen designs used or rejected because of performance considerations, which turn out to be bogus once somebody actually does some measurements on the real setup used for the application"

Martin Fowler, Patterns of Enterprise Application Architecture

Most applications today create or use data that need to be stored and retrieved. In the object oriented environment objects are usually used to represent data. These objects need to be stored or persisted. *Persistence* is "the characteristic of data that outlives the execution of the program that created it" [5]. *Object persistence* is the storage of objects that are available after program execution [92]. There are different *mechanisms* to manage these persistent data objects [80]. In Java these mechanisms are for example: object serialisation, Java Data Base Connectivity (JDBC), using EJB3 (Enterprise Java Beans) Entity beans [68], JDO (Java Data Objects) [93] or JPA (Java Persistence API) [68] to make objects persistent to a flat file or to a database. These mechanisms are sometimes called *persistence mechanisms*. These persistence mechanisms might store the data directly to a flat file on a physical disk or use data management software systems that manages and stores the data on the disk. Some well known data management software systems are: Object Oriented Database Management Systems (ODBMS's), Relational Database Management Systems (RDBMS's), Object-Relational Database Management Systems (ORDBMS's) and XML based databases. Database management systems (DBMS's) are classified based on the models they use. For example relational databases use a *relational model* and object databases use an *object model*.

A known problem that exists in the persistence of objects using a relational database is the mismatch between the object model and the relational model and the mismatch between the object programming language and the relational query language [53]. This mismatch is know as the *impedance mismatch*. To solve this problem object-relational databases have been created and traditional relational database vendors have included object persistence capabilities into their products. Object Relational Mapping (ORM) tools have been created to try and bridge this mismatch and to make persistence of objects easier for the developer. Today ORM tools are being used extensively with RDBMS's to persist objects. ORM tools, object-relational databases and relational databases with object storage capability can be seen as the hybrid solutions to store objects, as they use the *relational model* to store the object model. Object databases use and store the *object model* which is same as the application object model.

These databases are usually located on the bottom layer of an architecture design. Fowler et. al. [70] states that "layering is one of the most common techniques that software designers use to break apart a complicated software system". A common architectural style is the three-tier layered style as shown in Figure 1.1.

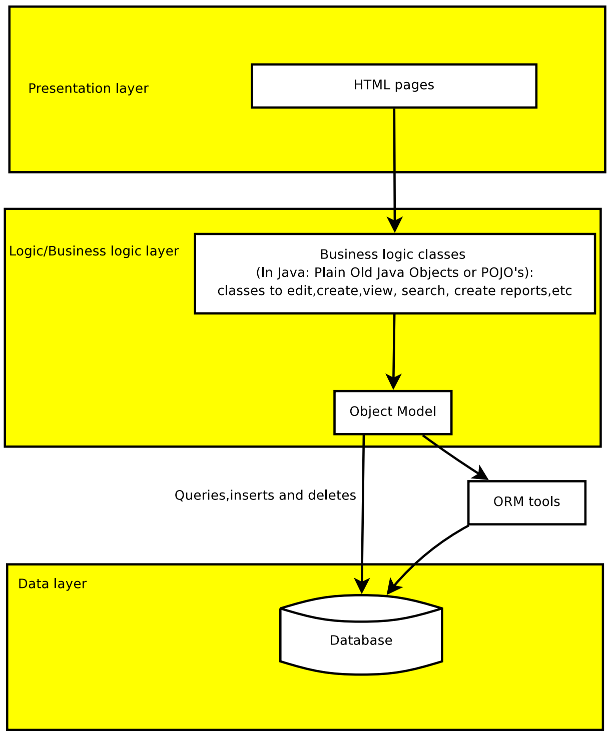

Presentation layer
HTML pages
Logic/Business logic layer
Business logic classes
(In Java: Plain Old Java Objects or POJO's):
classes to edit,create,view, search, create reports,etc
Object Model
Queries,inserts and deletes
ORM tools
Data layer
Database

Figure 1.1: Representation of a three-tier layered client-server style

Kruchten [86] defines an architectural style as: "a family of such systems in terms of a pattern of structural organisation". In the context of the three-tier architecture, the middle layer called the business logic layer, is where the business logic of a specific application is modelled as business logic objects or domain entities. Theses objects or entities are usually represented as objects using an object oriented programming language. The entities created in the business logic layer need to be stored for later retrieval if they represent persistent data. The entities are stored in the data layer usually using a database. For this study the focus is the data layer and specifically the *performance* of the databases and ORM tools used to persist objects in this data layer.

It is important to *choose the right* persistence mechanism and associated database for an application as it could influence the time to develop the application as well as the performance, scalability and maintainability of the application. A focus of architects is to optimise the performance of these layers and use the right tools and databases for the type of application under design or development. Looking at the data layer the architect or development team must decide between some of these possible persistence mechanisms and persistent storage facilities:

- A relational database, object relational database, an object database, flat files, or if using Java, object serialisation.
- An ORM tool to convert from objects to the relational database.
- Using EJB3 (Enterprise Java Beans) Entity beans, JDO (Java Data Objects) or JPA (Java Persistence API) to map Java objects to a database when using the Java language [74, 68, 93].

Each of these has some known advantages and disadvantages. For example traditionally object databases are seen as making development faster as the application object model and the database model is the same. While object databases assist with development, data mining and database reporting tools are not as mature or common as those found in the relational world. ORM tools are seen as creating a slight translation overhead but makes it possible to map objects to relational databases. ORM tools also assist with creating objects from legacy relational databases for application use. Relational databases are seen as very fast when doing sequential traversals while possibly slower with object traversals and joins between different tables. These advantages and disadvantages will be discussed further in the next chapter.

The study investigates and focuses on the *performance of object persistence* and will include recent ORM tools and compare them to object databases. This comparison is possible because both ORM tools and object databases persist the same unit namely objects. The ORM tools provide an extra layer between the business logic layer and the data layer. This study hypothesises that this extra layer and mapping that happens at that point, could slow down the performance of object persistence. The aim is to investigate this extra layer as well as the use of an object database, that cuts out this extra mapping layer. The study also investigates the impact of certain *optimisation techniques* on performance.

There have been other studies in this field. For example, Cattell and Skeen [52] investigated the performance of various relational databases against object databases, while Carey et al. [48] compared the performance of various object databases. More recently, Jordan [80] has made a study of all of Sun's persistence mechanisms. It was this latter study that inspired the idea of comparing the ORM tool approach of persisting objects to that of using an object database approach to persisting objects.

A benchmark will be used to compare ORM tools to object databases. The benchmark will provide us with criteria that will be used to compare them with each other. The particular benchmark we have chosen is OO7 [48], widely used to comprehensively test object persistence performance.

Because of its general popularity, reflected by the fact that most of the large persistence providers provide persistence for Java objects, it was decided to use Java objects and focus on *Java persistence*. A consequence of this decision is that the OO7 Benchmark, currently available in C++, has had to be re-implemented in Java as part of this study.

The Open Source Software (OSS) [13] movement is quite popular today and there are open source ORM tools, relational databases and object databases available. Proprietary and open source tools and databases are included in the study. The emergence of OSS adds another dimension to the comparative study. It also opens the possibility of investigating comparative performance of open source object databases against proprietary object databases. Indeed, one could also investigate open source ORM tools against proprietary ORM tools. However this comparison was not included in this study.

db4o [63] and Versant [15], representatives of object databases, and Hibernate [8], representative of the ORM stable, will be compared with each other. Hibernate and db4o are popular *open source* products while Versant is proprietary. The study will focus on running Versant, db4o and Hibernate with PostgreSQL in their *default modes* (out of the box) and use them as a normal programmer would. This initial out of the box state would then form a basis to which optimisation techniques will be added and compared.

As stated above *optimisation techniques* will also be investigated. This investigation will try and find how much certain techniques improve performance and in what circumstances could or should these techniques be used. The techniques that will be investigated are:

- indexes
- lazy and eager loading
- transactions
- caching
- different types of queries

1.1 Research objectives

The study has a few research objectives. These are to investigate the *performance* of object databases against that of ORM tools and how different optimisation techniques influence the performance. One of the research objectives is to understand *when to use* an object database or an ORM tool by investigating which of them perform better for different types of operations. The OO7 benchmark provides operations that involve traversals, queries, inserts and deletes. Using the results for the OO7 operations certain recommendations could be made. For example if an application performs many traversals of small object trees, the results of OO7 operations could be used to make a decision between using an object database or an ORM tool. The same is possible for queries. If an application is query intensive and performs *join*-like queries then OO7 benchmark results could be used to provide a recommendation. Although the benchmark results are used to provide recommendations, it is important to understand that a benchmark only provides a guide and that *real life applications* must be tested and benchmarked with the different options.

Part of the study is to investigate the OO7 benchmark in greater detail to get a clearer understanding of the OO7 benchmark code and inside workings thereof. While other studies have used OO7, the inside workings of OO7 have not been discussed in great detail. For example the number of objects that are created, traversed and queried by the OO7 benchmark is documented in *greater* detail in this study. Also the *process* to create and run the OO7 model and benchmark operations is discussed.

The performance of newer open source object databases are investigated and compared to that of older more established proprietary object databases.

Also the use of indexes, caches, eager and lazy loading are investigated to better understand what the impact of these techniques are.

The next section will discuss what type of research will be done.

1.2 Type of research

This research study is an *empirical* study following the *empirical method*. The empirical method can be defined as "relying or based solely on experiment and observation rather than theory" [16].

The study as stated before, will use a benchmark and try and show *observable facts*, in smaller non-real life experiments, following an empirical method. Using the *results* and *observations* it is hoped to gain some insight into object persistence performance of object databases, ORM tools and optimisation techniques and some plausible explanations for these results.

1.3 Layout of dissertation

This section describes the dissertation layout. Chapter 2 is an overview and a general background on object persistence, relational and object databases. Advantages and disadvantages of object and relational databases are also discussed in this chapter. Chapter 3 mentions benchmark classifications and criticisms against benchmarks. Chapter 3 investigates the OO7 benchmark in great detail. Its internal workings and some hidden pitfalls will be discussed. Chapter 4 is an overview of the Java implementation for the OO7 benchmark that was developed for the study, and discusses some of the details pertaining to the implementations created for db4o, Versant and Hibernate. In chapter 5 the *benchmark results* will be presented and discussed for each individual implementation. It will include the results for using different optimisation techniques. Chapter 6 will *compare the results* for db4o, Versant and Hibernate with each other directly. It includes scoring the overall performance of the respective tools. Chapter 7 discusses performance recommendations provided by vendors. The chapter also includes performance results when these recommendations are used. Chapter 8 will provide a summary, conclusion and reflections on this study.

Chapter 2

Persistence Background

"Thus, if you upgrade to a new version of your virtual machine, hardware, database, or almost anything else, you must redo your performance optimisations and make sure they're still helping"

Martin Fowler, Patterns of Enterprise Application Architecture

This chapter will provide background information about object databases, relational databases and ORM tools and discuss advantages and disadvantages of each. Definitions and background related to persistence, Java persistence and optimisation techniques such as clustering and lazy loading are discussed as well.

2.1 Object database management systems

Object database management systems (ODBMS's) are also called object databases (ODB), object-oriented databases (OODB) or persistent object stores. This section provides a few definitions as each provides a different insight into what an object database is.

Kim [84] defines an object database as follows:

"An object-oriented database is a collection of objects whose behaviour and state, and the relationships are defined in accordance with an object-oriented data model."

The Object Data Management Group (ODMG) [51], which defined standards for object databases defines

"a ODBMS to be a DBMS that integrates database capabilities with object-oriented programming language capabilities" [51].

Ambler [30] defines an object database to be a

> "persistence mechanism that stores objects, including both their attributes and their methods".

From these definitions it is clear that an object database is based on an object oriented model and language and stores the whole object with its *relationships* and object *methods* together in the database.

As stated, object databases are based on object-oriented theory and they use object models. The object model has the following features: object and object identifier, attributes and methods, encapsulation and message passing, a class, type and class hierarchy, inheritance, polymorphism and complex objects[84, 69, 41, 65, 83].

The next sections will discuss the history and provided functionality of object databases.

2.1.1 History of object databases

Object databases became available during the middle 1980's and early 1990's and they were mainly used in Computer-Aided Design (CAD) and Computer-Aided Software Engineering (CASE) tools. Bertino and Martino[41] classify these early systems into three generations.

The *first generation* systems include: G-Base by Grapheal; GemStone by Servio Corp in 1987; Vbase by Ontologic; etc. They were characterised as standalone systems.

The *second generation* systems included: ONTOS by Ontologic in 1989; ObjectStore by Object Design in 1990; Objectivity DB by Objectivity in 1990; and Versant Object Technology also in 1990. These systems were mostly based on the client/server architectural model and ran on Unix operating systems.

Itasca was the first *third generation* system. Bertino and Martino[41] state that Itasca was "the commercial version of the ORION project from Microelectronics and Computer Corporation (MCC)". The other well know third generation product was O2 by Altair and Zeitgeist and developed by Texas Instruments in 1991. Poet was also developed in 1992 by NKS Software and was well known during the 1990's. The third generation systems were more advanced and provided more management features and data manipulation languages[41]. The dates of when these systems came out were obtained from Kroha [85].

In the last few years *open source* object databases such as db4o and ozone[4] have appeared. db4o is an object database as well as an embedded database. "An embedded database system is a DBMS that is tightly integrated with an application that requires access to stored data, such that the database system is “hidden” from the application’s end-user and requires little or no ongoing maintenance”. [18].

Most of today’s object databases have adopted the ODMG 3 [51] and JDO [93] standards. Currently a new "4th generation" standard is being developed by the OMG (Object Management Group) [11] along with inputs from object database vendors [12]. The last few years have also seen the establishment of the non-profit ODBMS.ORG portal with a focus on providing a forum with resources and discussions on object databases [12].

2.1.2 Functionality provided by object databases

Object databases provides similar data management features and functionality to relational databases. Some of the functionality and features are given below [84, 82, 65, 83]:

- Persistence
- Transaction management
- Recovery and backup management
- Concurrency control
- Versioning and schema management
- Indexing
- Query processing, optimisation and object query languages
- Pointer swizzling
- Clustering of objects
- Optimisation features, performance management and monitoring
- Cache management
- Visual interfaces
- Security, authorisation and authentication

Concepts appropriate and relevant to this study will be discussed in the following subsections.

2.1.2.1 Persistence

Persistence is "the characteristic of data that outlives the execution of the program that created it" [5]. *Object persistence* is the storage of objects that are available after program execution [92]. For this study the focus is on Java object persistence.

There are different methods to make objects persistent *using an object database.* Java objects can be made persistent in several ways [51, 40, 100]. Using a *post-processor*, the byte code of the class files are modified, adding in the persistence capabilities; using a *pre-processor*, the Java source files are modified before compilation; the persistence can be embedded in the VM using a *modified virtual machine* (VM), or the Plain Old Java Objects (also known as POJOs) are persisted, without any need for modifications, just by calling a method or by extending a certain interface to mark the object as persistent [53]. Objects can also be made persistent by reachability from other "persistent root objects" [53] as explained below.

A post-processor is also called a static process [40] and is *transparent* to the application code. This means that that the developer does not need to modify the code during development to add this persistence code and that the *persistence code that is added is hidden* from the developer. Versant uses a post-processor.

A modified VM technique involves the rewrite of a VM with the VM modified to include object persistence and is known as active process [40]. This technique is also transparent as no code is modified to include persistence. In the PJama study [33] the Java VM is modified to include persistence. Gemstone Facets [71], an Java object database, is based on a modified virtual machine to provide persistence. One possible risk with the modified VM is that it is modified and built on a specific VM and JDK (Java Development Kit). If a new JDK comes out, and development is done in this new JDK, the application code could implement new features in the JDK that are not in the VM and this could cause a mismatch and conflicts at run-time.

Atkinson [36] points out that "if an object is persistent, all of the objects that it references must be made persistent". This concept is called *persistence by reachability* or transitive persistence [51, 36, 33].

Object databases can have *root objects* [40, 53]. Certain objects can be marked as a root objects. From the root object, other persistent objects that are reachable can be traversed. If objects are added or linked to the root object they have to be persisted as well. This persistence is required because of *persistence by reachability.*

Another important concept with regard to persistence has to do with an object's states. An object can be in a *persistent state* or a *transient state*

[51, 40]. An object is said to be in a *persistence state* if the object is already stored in the database and can be retrieved later. An object is in a *transient state* if that object is not persistent at this point in time but could later be persisted. Other states, such as states of references to objects, are discussed in further detail in [40, 93].

Also related to persistence is the concept of transparent persistence [33, 35]. Moss and Hosking [34] refers to the *transparency* of a program that arises when "a program that manipulates persistent (or potentially persistent) objects looks *little* different from a program concerned only with transient objects". Transparent persistence is about how *transparently* persistent data is stored and accessed in the underlying database and about making persistence easier and almost hidden from developers. ORM tools, as well as object databases strive to achieve transparent persistence.

The PJama (or PJava) [33] research project also investigated Java persistence and focused on *orthogonal persistence*. "Orthogonal persistence is the provision of persistence for all data irrespective of their type". Jordan [80] compared the performance of Sun's Java persistence mechanisms which included PJama using a modified OO7 benchmark. This is the same benchmark that will be used during this study. The OO7 benchmark will be discussed in greater detail in the next chapter.

2.1.2.2 Transaction management

Kim [84] defines a transaction as "a sequence of reads and writes against a database". He notes that a transaction has two properties called atomicity and serializability. Kemper and Moerkotte [82] also state that transactions must have the ACID properties:

- Atomicity: all the actions (reads and writes) of a transaction must succeed or all of them must be rolled back.

- Consistency: The transaction takes "the database from one consistent state into another consistent state".

- Isolation: "There should be no interference among different transactions accessing the database concurrently".

- Durability: "The effects of a completed transaction remain persistent even if the database" crashes.

Serializability "means that the effect of concurrent execution of more than one transaction is the same as that of executing the same set of transactions one at a time" [84]. Transaction management is there to ensure that the ACID properties are adhered to.

Amongst other things this study is interested in the *performance difference* between grouping read and write actions into one large transaction on the one hand and grouping them into smaller transactions, on the other.

2.1.2.3 Versioning and schema management

Kemper and Moerkotte [82] define

> "a *schema* as a set of type definitions–including their definitions of the structure and the behaviour".

An schema changes as objects are added and deleted from the schema over time and this is called schema evolution [82]. An object or a schema can be versioned. Kim [84] defines schema versioning as

> "the versioning of a single logical schema".

This versioning of the schema makes it possible to load a specific version of the objects into memory for use. Loomis [89] mentions that each object database has a schema manager and that it functions as the "link between the database and programming language environments".

2.1.2.4 Pointer swizzling and lazy and eager loading

Loomis [89] states that

> "Swizzling is a technique the object DBMS uses to change object references to main memory pointers".

Kim [84] mentions that there is a mapping of objects from their in-memory format to their disk format in the database and that references between objects "need to be converted between absolute addresses in-memory and relative addresses on disk" [84]. These object references are pointers to other objects that might not currently be in-memory and are still on physical storage. Swizzling is used when objects are loaded into main memory, virtual memory or an in-memory cache. Kemper and Moerkotte [82] define three dimensions of pointer swizzling: in place/copy, eager/lazy and direct/indirect. This study is interested in *eager and lazy loading* which is very similar to swizzling.

Under eager loading, when a given object is loaded into memory, all other objects which are referenced by this object are also loaded into memory.

Under lazy loading, the loading into memory of these referenced objects is delayed until they are actually needed.

Tests have been created for this study to investigate the impact of lazy and eager loading on performance.

2.1.2.5 Clustering of objects

Kim [84] defines clustering as a

> "technique used to store a group of objects physically close together so that they may be retrieved efficiently".

Clustering techniques are used to keep page faults to a minimum and *locality of reference* high. By locality of reference we mean the degree to which objects that are clustered together are related to each other. Whether this clustering be on disk, in memory or in the cache. Some of the clustering techniques are to group together, on the same page or segment, objects of the same class type or objects that are related or connected with each other.

2.1.2.6 Cache management

Cache, or buffer management is about providing and managing a buffer for "accessing objects in main memory after they have been fetched from disk" [53]. By caching *frequently* accessed objects, or objects that have a high *locality of reference*, the access speed of these objects is often increased because there is no need to fetch these objects again from disk. Access speeds are increased even more if the whole database or object model can be cached in memory.

2.1.2.7 Indexing

An index is an search key that is generated for a field of an object. The key helps to quickly retrieve the object during a query using that field. Usually the key, and the reference to the location of the object, are stored in a index table. The index table could be implemented as a B-tree or hash table for example. If new objects are stored in the database, and they have fields that are being indexed, an entry has to be added to the index table as well.

2.1.2.8 Query processing, optimisation and object query languages

The execution of a query happens in two stages according to Kim [84]. Stage one is to optimise the query and stage two is to actually run the query on the database using the optimal query created in the first stage.

A query language can be used to obtain information from the database model. Most relational databases have three languages: the query language (QL) of which SQL (Structured Query Language) is the most well-known, the data definition language (DDL) and the data manipulation language (DML) [69]. A DDL is used to define the data definition and schema while the DML is used to manipulate the data, for example, by inserting data. Object databases based on the ODMG standard have equivalent languages called the object definition language (ODL), the object query language (OQL) and the object manipulation language (OML) [51]. The ODL is defined by the ODMG to be a specification language that is used to specify object types. These specifications must conform to the ODMG Object Model to be compliant with the ODMG standard. The OQL is a query language for objects and can return properties of an object or whole objects. The OML is defined by the ODMG to be a language for retrieving objects and modifying them.

This study will rely on various OQL's. These are Versant's query language called VQL [100], Hibernate's HQL [76], db4o's SODA query API [61].

2.1.2.9 Security, authorisation, authentication

The security functionality of an object database refers to the way in which access to data is protected. This includes the authentication process to verify the identity of the user who is trying to access the data. An authorisation process is also required to make sure that a user who is trying to access data has the required access rights to that data. These control processes are provided by the database management system.

2.1.3 Advantages and disadvantages of object databases

Object databases provide the following advantages:

- Object databases can model and store complex objects and relationships using the object model. Object databases are able to store whole collections such as lists, sets and map on disk.
- Code and data are stored together and managed together. This means that there is no need to break op the manipulation code from the model.
- Object oriented programming languages such as Java and C++ are used as the data manipulation and query languages, so no new language such as SQL needs to be learnt and used. This provides a solution to the impedance mismatch problem.

- Only one model is required. The model defined in the object oriented programming languages is also the data model of the database. There is no need to map the object model to the relational model. Using one object model can result in shorter development time and better programmer productivity as no mapping is needed [89].
- Objects and the references (pointers) to their related objects are stored together and this will generally improve search or traversal time as no joins at run-time are needed [89, 40]. This is dependant on how the objects and their relations are stored on disk. Pointer swizzling is probably needed here.

Some of the disadvantages of object databases include:

- Users want access to the data for running reports on the database. SQL is used to query and run reports on relational databases and when using a object database they either have to learn OQL or get developers to write reports using programming languages and extra libraries such as Jasper Reports [21]. While there are many graphical reporting tools available that run on relational databases, there aren't many that run on object databases.
- An object database is seen has having "no formal semantics", whereas relational databases are based on the relational set theory in mathematics [41, 44].
- Some of the early object databases were seen as *slow*. While performance was a valid concern in the late 80's and early 90's it is generally not a relevant reason today. The OO1 benchmark study by Cattell and Skeen [52] has shown that object databases often perform just as well or better than relational databases. Case reports have shown that object database often perform well [58]. Chaundhri and Loomis [55] and Chaundhri and Zicari [57] provide evidence of case reports, research studies and benchmarks which show that object databases do perform well and in diverse application areas.
- Lack of a standard. While the ODMG and JDO standards have provided standards for use with object databases, these are not so well integrated or used by many vendors. The new JPA standard for Java persistence does not even mention object databases.

2.2 Relational databases

A relational database is a database based on the relational data model. The relational model is based on the "set-theoretic notion of relation" [82] and a relation is "a subset of the Cartesian product of various domains" [65]. A relational database management system (RDBMS) has functionality similar to an object database. Some of the functionality includes: transaction management, recovery and backup management, concurrency control, schema management, indexing, query processing and query languages, optimisation features and performance management and monitoring, cache management, visual interfaces, security, authorisation and authentication. Object databases manipulate and store objects whereas relational databases operate on records in tables. Relational databases will not be discussed in-depth as the main focus of the study is on object persistence through the use of ORM tools on relational databases.

2.2.1 Advantages and disadvantages of relational databases

Relational databases have been around for a long time and are currently the most widely used database technology. Delobel et al. [65] list some of the advantages provided by relational databases:

- Simplicity: Relational databases and their schemas are reasonably simple and intuitive.
- A good theoretical base: Relational databases are built on relational and set theory.
- Data independence: The data and the programs are separate and any programming language can get access to the data using SQL. This is an important advantage for many companies and developers. Historically, one of the reasons why this was an important advantage was that different languages could not communicate with each other. Data independence was used as a way to communicate and share data with other companies and developers if they were using different languages. Today new technologies exist to share data and communicate between different systems and languages. Some of these technologies are the Common Object Request Broker Architecture (CORBA) which allows developers to share objects and functionality with other languages in a distributed environment. Web Services and XML are used today to exchange data and call different services.

- Advanced integrity and security: The database enforces integrity constraints and provides security on many levels: database, table, etc.
- Manipulation of data in sets: Delobel et al. [65] state that "in contrast to first-generation systems, the relational model provides data manipulation languages that allow sets to be manipulated globally".

While relational databases are widely used they do have some disadvantages [65, 40]:

- Over simplified model: The model was constructed for simple data structures and relationships. It assumed the use of atomic types such as integers and strings. These days with the use of object languages the data is much more complicated with many relationships to other objects. This is because object oriented languages have made it possible to model the real world with objects and their relationships. It is possible to map these relationships in the relation model, but as Delobel et al. [65] and Loomis [89] state this is not easy to do and the semantic data of these relations may be lost. To map these relationships one has to use primary and foreign keys and joins of tables. Joins usually require extra processing time.
- Limited manipulation languages: SQL languages are tailored for querying. They have a limited set of operations and can be seen as relationally complete but not computationally complete [44]. Usually SQL is embedded in programming languages to provide persistence storage and retrieval operations on the data. This means that the programming language has to convert its types to those of the underlying relational database. This gap between the programming language's types and the relational database types is known as an impedance mismatch. The impedance mismatch and the embedding of SQL code in programming languages force developers to think in two worlds. Not all developers are experts in SQL and this can cause problems during development and extra time to map programming language types to relational types [40]. ORM tools such as Hibernate make the mapping of primitive types to database types easier.
- Runtime checking: SQL is only checked at runtime which can make debugging and testing of systems more complex [40].
- Cost of joins and speed when traversing relationships: Loomis [89] mentions that in most cases, retrieval of objects with relationships involve

relational selection and joining. If there is order involved in the relationships then the relational data in the selects or joins must be sorted as well. Loomis [89] further states that joining a table with *n* rows with a table of *m* rows requires *n*m* comparisons. Optimisations can be made in the relational world for these types of queries. Most object databases store the relationships and references in the database as pointers and this will generally increase retrieval and traversal speeds.

2.3 Object Relational Mapping tools

To resolve the impedance mismatch problem, various hybrid solutions have been proposed. Thus, object-relational databases have been developed, and traditional relational database vendors have included object persistence capabilities into their products. Object Relational Mapping (ORM or also known as O-R) tools have also been developed in an attempt to bridge this mismatch and to make persistence of objects easier for the developer. These tools provide a mapping between the object model and the relational model, acting as an intermediary between an object oriented code base, and a relational database. Bauer and King [39] defines the Hibernate Java ORM tool as "the automated (and transparent) persistence of objects in a Java application to the tables in a relational database, using meta-data that describes the mapping between the objects and the database. ORM, in essence, works by (reversibly) transforming data from one representation to another".

Hibernate and Toplink (from Oracle) are examples of ORM tools. Hibernate is increasingly being used and integrated into products such as JBoss, which is a Java EE (Enterprise Edition) application server.

2.3.1 Advantages and disadvantages of ORM tools

As early as 1990 it was predicted that in the future relational databases will become more like object-relational databases and that companies will be mapping their objects to relational databases [84]. Ambler made similar predictions in 1998 [30]. Looking at the database market today it would seem that this has occurred. Looking at the growth of ORM tools and considering that companies such as Oracle have bought the Toplink mapping tool, it would seem that companies and developers have not moved over to object databases. Some of the reasons why companies are using ORM tools include:

- Legacy data in a relational database: Barry [37] states that "the data already resides in one or more relational databases".

- Political or technical reasons: Barry [37] states that “the data is new but for political or technical reasons they are using relational databases.“
- Knowledge of developers: Most students and new developers are taught about relational databases.

ORM tools provide the following advantages:

- Developers can work with *objects* and the mapping tools will transparently persist the data.
- The programming code is reduced in size as the developer does not need to write all the mapping code or data mapping layers and SQL statements. The development time is therefore reduced [39].
- The maintainability of the code is improved because object-relational mapping involves “fewer lines of code” and because changes to the object and relational models are easier [39].
- It takes away the need to embed SQL code all over the code base.
- Vendor independence [39, page 28]. “An ORM abstracts your application away from the underlying SQL database and SQL dialect” [39]. Many ORM tools support different underlying databases and makes it possible to switch between different relational database vendors.
- Barry [38] mentions that the ORM mapping tools provide caching mechanisms and that the use of caching can reduce the impedance mismatch and improve performance.

The following disadvantages and difficulties are associated with ORM tools:

- It is not trivial to map one-to-many relationships, many-to-many relationships, and inheritance structures, especially when the application is large and changing, and the mapping is manual.
- Mapping still requires fairly intimate knowledge of both the object and relational models. Bauer and King [39] state that they “believe that Java developers must have a sufficient level of familiarity with—and appreciation of—relational modelling and SQL in order to work with ORM. ORM is an advanced technique to be used by developers who have already done it the hard way”.
- Even though ORM tools ease object mapping many developers are still embedding SQL-like code in their code and thus it is still not a pure object solution using just one programming language.

- ORM tools have still not solved the impedance mismatch problem.
- OQL based languages like HQL still suffer from no compile time checking. If for example the class name is changed in an object model the OQL/HQL statements in the code, must also be changed. The OQL statement is a string so if the class is incorrectly renamed an error will only be thrown at run time. Object databases using OQL will have a similar problem. New efforts such as Native Queries [59] in db4o and LINQ [10, 43] write queries in the programming language that can be compile time checked.
- There is a performance overhead for mapping between objects and database tables. Bauer and King [39] state that because ORM is "implemented as middleware" there exists ways to improve and optimise the performance of this extra layer. This study will investigate if this extra layer causes a performance overhead.

The next chapter will discuss the OO7 benchmark.

Chapter 3

OO7 Benchmark

"The choice is always the same. You can make your model more complex and more faithful to reality, or you can make it simpler and easier to handle. Only the most naive scientist believes that the perfect model is the one that perfectly represents reality. Such a model would have the same drawbacks as a map depicting every park, every street, every building, every tree, every pothole, every inhabitant, and every map."
James Gleick, Chaos

Questions about the performance of *object databases* arose during the late 80's and early 90's, soon after academic and commercial object databases were becoming available. It was at this time that several benchmarks were created to measure their performance. Well known benchmarks included HyperModel [32], OO1 [52] and OO7 [48]. The OO1 benchmark was intended to study the performance of engineering applications. The HyperModel approach was based on earlier versions of the OO1 benchmark. It incorporated a more complex model with more complex relationships and a wider variety of operations. The HyperModel benchmark focused on the hypertext model. The OO7 benchmark was based on both of these benchmarking efforts. This chapter will provide an overview of the OO7 benchmark and also discuss why this benchmark was selected for our benchmarking and comparison investigation.

3.1 OO7 benchmark overview

OO7 was designed to investigate various features of performance, and it included complex objects which were missing from the OO1 and HyperModel benchmarks [48]. While the earlier benchmarks used single value results, the

OO7 benchmark provided a collection of results. Mugal [91] also suggests that a single valued benchmark result is not as useful as having a collection of results. Carey et al. [48] indicate that the benchmark is intended to investigate

> "associative operations, sparse vs dense traversals, updates to indexed vs. non-indexed object attributes..."

The benchmark uses a *hierarchy of objects*, modelling an engineering design library. Each `Model` contains either `BaseAssembly` or `ComplexAssembly` objects and has an associated `Manual` object. `ComplexAssembly` objects contain other `BaseAssembly` objects. `BaseAssembly` objects contain `CompositePart` objects, that contain `AtomicPart` objects. `AtomicPart` objects are connected with each other through `Connection` objects. All of these objects have some basic attributes and some collections representing one-to-one, one-to-many and many-to-many relationships. Most of these relationships are bi-directional. See Figure 3.1 which is based on the one in [46] for more details.

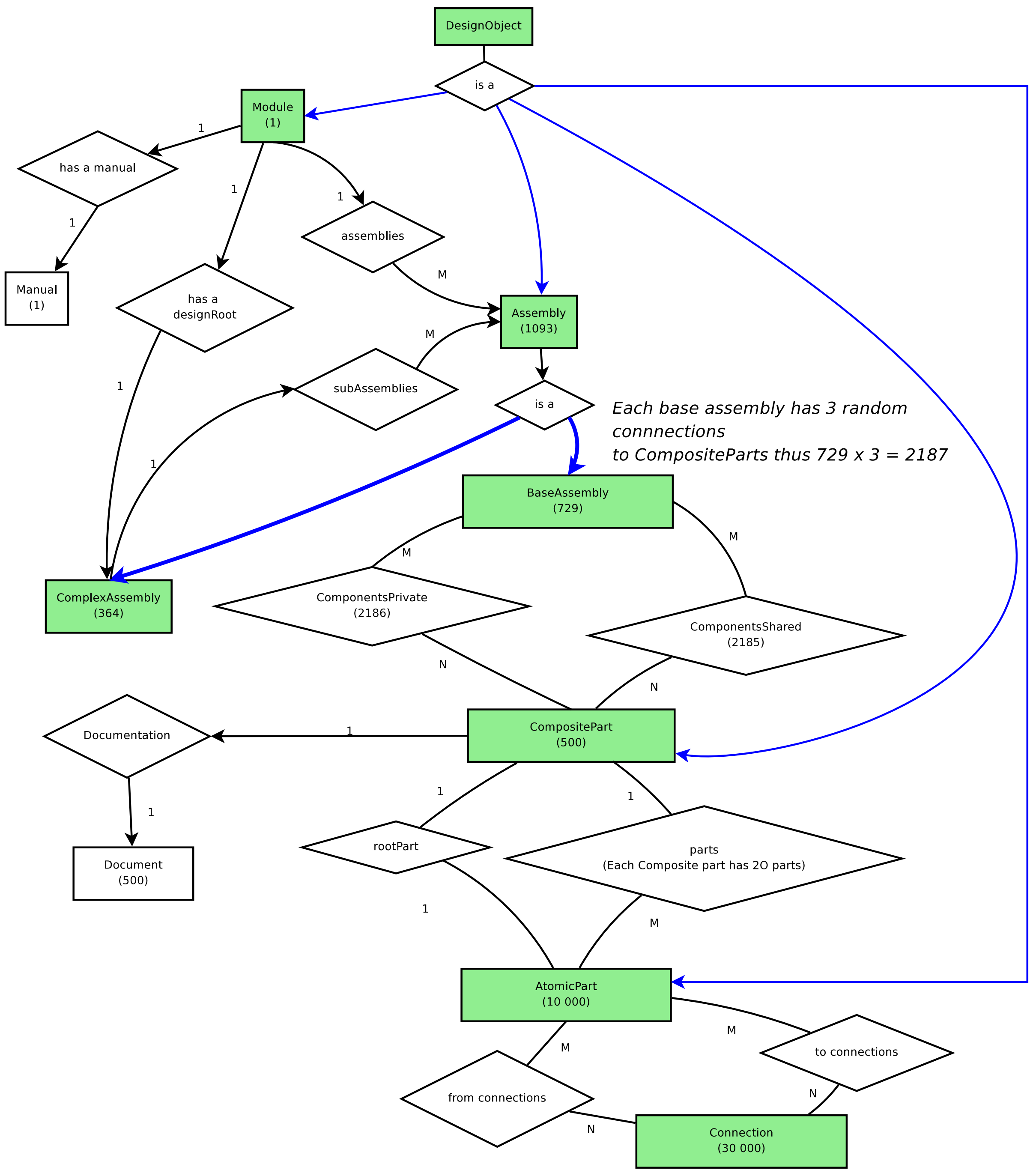


Figure 3.1: OO7 object model and the number of objects created in the *small* configuration

The object model is used as the target of various navigation and persistence operations. There are configuration settings that can be changed to influence the number of objects created, the number of models, the number of connections, etc. The benchmark also has a *small*, *medium* and a *large* database configuration which specifies the number of objects to be created in the benchmark. For each of these database configurations it is possible to configure the number of connections between objects. An example of this is where an `AtomicPart` is connected to other `AtomicParts`. Through the use of the configuration value, it is possible to control the number of these connections [48]. Carey et al. [48, 46] defined 3, 6 and 9 as suggested connection values. In this way, it is possible to have a small configuration with 3 connections, a small configuration with 6 connections, a medium configuration with 9 connections, etc.

The benchmark contains operations that operate on the design library and can be grouped into three categories: *traversals*, *queries* and *modifications*. Although these are well-documented in [46], various features of these operations that are not so obvious will be mentioned below. The modification operations refer to the insertion and deletion of objects, and to some of the traversal operations that not only traverse the object hierarchy but also modify the objects by swapping values, renaming them, etc. Traversals are performed "by selecting a random part and then performing a seven-level depth-first traversal (*with multiple visits allowed*) of the parts reachable from there" [45].

Operations are run as cold or hot [48]. Cold runs are runs where all the caches should be empty, and hot runs are where caches are full. There is also the in-between notion of a warm run, which refers to an initial cold run to fill up the caches, followed by a hot run. Chapter 5 will provide the measurements in regard to these operations under both cold and hot running conditions.

One of the early difficulties that arose in attempting to use the OO7 benchmark was the issue of query languages [47]. Not all of the systems tested then had query languages, and those that had, had languages that differed in capabilities. For those that did not have query language capabilities, C++ query methods were written in the hope of enabling comparison across all of the systems. Carey et al. [47] conceded that this could favour some implementations with no query language capabilities.

This possible bias is avoided in the present study, since all the systems that have been chosen for this study have their own object query facilities. db4o uses SODA (Simple Object Database Access) as well as Native Queries, Hibernate uses HQL (Hibernate Query Language) and Criteria API, and Versant uses VQL.

In selecting a benchmark for comparing our chosen target systems (persistence mechanisms and persistent stores), the OO7 was a natural prime candidate. Because it has a deep object hierarchy with many complex relationships (one-to-many and many-to-many), it is well-suited to assessing non-trivial object oriented applications. Furthermore, the fact that the code was available in C++ meant that it could be converted to Java with relative ease. Finally, this study was influenced by the fact that OO7 had been used in the recent past for a research study into persisting Java objects [80]. While our study is similar to the Jordan [80] study, they left out the OO7 queries. Our study includes queries.

Other studies using OO7 are [72, 67, 87, 101, 90, 73, 60]. Some of these have a Java version of OO7 but have not documented their approach of conversion from C++ to Java and for most the code is not freely/easily available. These studies also have a different focus from our study. Some of them investigate concurrency, software transactional memory, issues with regards to PJama, etc.

In the next section benchmark classifications are discussed.

3.1.1 Benchmark Classifications

In this section the different classifications for benchmarks are discussed and the OO7 benchmark is then classified according to them.

The following type of benchmarks are defined by [91, 102, 81]:

- Official benchmarks: Benchmarks that will be widely used. There are two sub-categories:
  - vendor benchmarks which are "intended to prevent atrocities in the marketplace".
  - and research benchmarks which are "intended to further the scientific process".
- Real user usage benchmarks: Benchmarks based on real users usage, through the use of traces, or a real system that is being tested. This benchmark is also called a real or realistic benchmark. Realistic benchmarks are difficult to repeat.
- Micro benchmarks: These are used to benchmark *components* of a system and not the overall performance of a real system. Micro benchmarks provide more control because only a small part of the system is being tested and they are more *repeatable*. This benchmark as well

as synthetic benchmarks can be oversimplified and might not be very good at predicting real production system performance [91, 81].

- Run a bunch of programs: DaCapo [42] benchmark is an example of such a benchmark.

- Synthetic benchmarks: These benchmarks create synthetic or artificial data loads for testing. Synthetic benchmarks are more repeatable and they are easier to modify. For example it is easy to modify OO7 to add new implementations for other persistence mechanisms and persistent stores, and compare them against each other. Synthetic benchmarks should be considered as providing information of how a system might perform.

- Macro benchmarks: These benchmarks are more complex because they test or simulate the real life system. They can become large and difficult to control.

These classifications can be combined. A benchmark that can create a synthetic load to test small components of a system would be classified as a synthetic micro benchmark.

Using the definitions above, the OO7 benchmark is classified as a *research, micro benchmark* with a *synthetic* load that tries to simulate real world loads and operations.

3.1.2 OO7 short comings and critique

Mogul [91] states that it is important that benchmarks should be *repeatable*, *relevant*, use *realistic* metrics, be *comparable* and widely used. OO7 was selected for this study as it complied with the above criteria. OO7 is repeatable, is relevant to object persistence performance testing, it uses realistic metrics for testing components of a persistence mechanism and persistent store, and has been used in other studies.

Although there are other studies that have used OO7, it will be difficult to compare our results against theirs. Some of the studies like the original OO7 study [48, 46] used older versions of programming languages and were mainly implemented in C++. Our study did try to update the C++ version of OO7 so that it could compare the Java and C++ versions of the Versant database against each other. This effort was abandoned because of time constraints and will be investigated later. Other newer studies like [78] which is also based on a Java version of OO7, removed tests that modified the model, as this was not part of their focus, whereas our study includes them. Although it is difficult to compare to other studies, it should be easy to take our implementation and add other persistence mechanisms and persistent stores, and compare them to our results. To be comparable one must be testing for the same or similar concept [95].

It is important to understand the critique and possible short comings of OO7. Kakkad [81] states that there is normal program behaviour, which is more CPU intensive, and database program behaviour which is more I/O intensive. They argue that OO7 and OO1 are better suited for database behaviour testing and not normal program behaviour testing. It is also important to include the hot times and not warm traversal times because some systems might prefetch objects into the cache sooner than other systems. OO7 reports on the hot times, and our study is focused on testing the persistent store (database) or persistence mechanism behaviour.

According to Kakkad [81] the OO1 benchmark and possibly OO7, have a *low locality of reference* and that this could influence techniques used in pointer traversals, pointer dereferences and clustering. This could be a result of the random linking of objects in OO1 and OO7. Our study is focused on the performance of traversals, traversals with modifications and queries and not so much on the investigation of how these persistence mechanisms and persistent stores, achieve this performance. Our study does not investigate the impact of clustering. OO7 creates a model and asks the persistence mechanism to persist this model. If the persistent store, under the hood, is able to cluster these objects together to get a better locality of reference then so be it. Other benchmarks, like ACOB [66] were created to investigate clustering and different database server architectures and would be better suited for studying clustering and locality of references.

It is unknown whether real life systems show a high degree of reference locality and good clustering. Consider a financial or health system as an example. As time passes more transactions, accounts and products are linked to user accounts. Locality and clustering could be influenced in this example by the following possibilities:

- New products can be added a few years later and because of space issues not added to the same extent or page.

- Users can stop using a certain product or the product might not be offered anymore. This could imply that the product could be deleted over time and create open space on the extent.

- Products can be offered by other divisions with their own servers and products. This product could be linked to a users account but actually live on another separate server. This means that not all the data are clustered together on the same extent or even disk.

The example shows that even real life systems will generally start out with a high degree of reference locality and good clustering and then after a few years the clustering and reference locality might degrade. The study will not investigate clustering and locality of reference as it is not part of the main focus.

In other studies it was found that OO7 is not well suited for a CAD workload as it was not based on a real CAD application or traces of CAD applications but based on a synthetic load [81, 95]. Chaundhri [102] also states that as the model tried to simulate a CAD application that it could not be reliably used to test real life commercial and financial systems. OO7 was selected for this study not for the fact that it has a CAD-like model, but rather for the fact that it has a good, deep, tree like, hierarchical object model. It includes inheritance, bi-directional associations and one-to-one, one-to-many, and many-to-many associations. These features seem to represent object oriented models used today. It is important to remember that OO7 is a synthetic benchmark that will be used to test the performance of some features (traversal speed, query speed) of a persistence mechanism and persistent store.

Another critique against OO1 and OO7 is with regards to the collection types used in the benchmark. It is stated that they do not stipulate what "kinds of a containers" must be used for the collections. Bags, sets or lists are different kinds of containers. The critique is not true as Carey et al. [48, 46] does provide an OO7 benchmark class schema in their appendix in which they specify what kind of "containers" should be used for the collections. While OO7 does specify that a *Set* or *List* must be used for a collection, the persistence mechanism and the programming language can provide its own implementation of a set. For example in Java there are different kinds of sets: SortedSet, HashSet and TreeSet. The implementer of the benchmark can select any of them.

Nothing stops an implementer from using the persistence mechanism's or persistent store's own implementations of collections. Kakkad [81] states that this makes comparisons difficult if persistence mechanisms use different collections. For this study the use of different collections did not matter. If a persistence mechanism or persistent store did provide a better and faster collection it was used and documented. Hibernate and db4o use the standard Java collections whereas Versant provided their own persistent collections (VVector, VHashtable, etc.) [98] and ODMG collections. db4o is busy creating new advanced fast collections.

Kakkad [81] mentions that the random number generator implementation could differ between different operating systems and that this might influence the performance results. The study has found that the random number generator in Java differs from the one in C++. This difference in the random number generator influences the number of objects that are traversed and queried but the same number of objects are still created in all the tests. Section 3.3 will provide a discussion on the use of the random numbers and the random number generator.

Chaundhri [56] critiques the databases sizes used in OO7 studies and states that larger data sets are needed. Most studies using OO7 either use the small or the medium database. This study agrees that the database sizes used are too small compared to the large databases that exist today. Even though the OO7 small database is very small this study still used it because it provides a quick initial insight into the performance of the persistent mechanisms and persistent stores. The OO7 small database results can then be compared to the large database sizes. This means that it is possible to see if a persistent store performs better for a small or large database. The use of the large OO7 data sets was investigated, but it was found that it needs more processing power and memory to run than was available for the study. Nevertheless the large OO7 data sets will be discussed in later chapters.

For further critique on benchmarks read Chaundhri [56]. Chaundhri [54] also provides a summary of other benchmarks related to object databases.

The following sections will discuss how the OO7 model is created and some of the interesting issues found in OO7.

3.2 OO7 model creation

This section describes how the OO7 model is created during the benchmark creation process. The creation process was not well documented and mostly embedded in the code.

The creation process consists of two phases. Figure 3.2 provides a visual representation of the two phases.

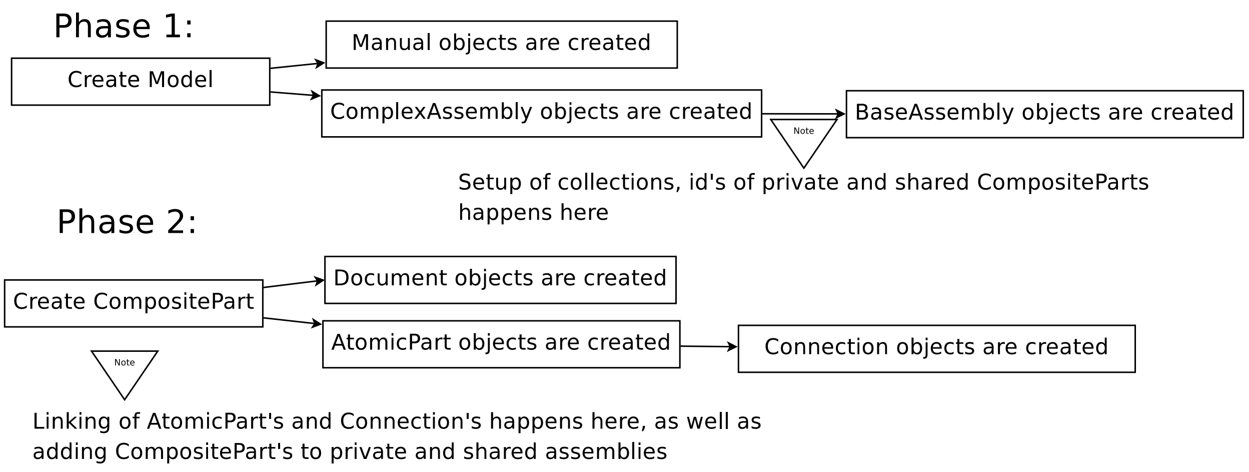

Phase 1:
Create Model
Manual objects are created
ComplexAssembly objects are created
BaseAssembly objects are created
Note
Setup of collections, id's of private and shared CompositeParts happens here
Phase 2:
Create CompositePart
Document objects are created
AtomicPart objects are created
Connection objects are created
Note
Linking of AtomicPart's and Connection's happens here, as well as adding CompositePart's to private and shared assemblies

Figure 3.2: Phases of OO7 model creation

In the *first phase* the *Model*, `Manual` and `ComplexAssembly` objects are created. The `ComplexAssembly` objects in turn create `BaseAssembly` objects which are added to the current`ComplexAssembly`'s *subAssembly* collection. `BaseAssembly` objects contain two collections, the *private* and *shared* collections. These collections will, after phase 2, contain `CompositePart` objects. In phase one, when `BaseAssembly` objects are created, random `CompositePart` ids are created. These `CompositePart` ids are generated by the *random number generator* and stored in a two temporary maps. One map represents private collections and the other the shared collections. These two collections are contained in the `LinkingMap` class.

Each map contains a `<CompositePartId, BaIdList>` pair where the `CompositePart` *id* is a key and the associated value is a list of id's, called a `BaIdList`, that reference the base assemblies using this associated `CompositePart`. The `BaIdList` list is created by taking the *current* `BaseAssembly` id and adding it to this list for the current `CompositePart` id that was generated by the random number generator.

These two maps, and the id's that it contains, will be used in the second phase to connect and set the *private* and *shared* relationships between `BaseAssembly` and `CompositeParts`. It is important to understand the ids are used because the `CompositePart` objects have not been created as yet.

The private and shared collections of relationships track which `CompositePart` each `BaseAssembly` uses and vice versa. These two maps in the `LinkingMap` class are stored in memory.

In the *second phase* the `CompositePart` objects are created which involves the creation of the associated `Document`, `AtomicPart` and `Connection` objects. The `AtomicPart` objects are linked (associated) with each other using `Connection` objects. Each atomic part can play a "from" or a "to" role in the association. These linkages happen by iterating through `AtomicParts` and selecting the "to"-part to connect with *randomly*.

The *private* and *shared* collections that were created in the first phase are now used to connect `CompositeParts` to `BaseAssemblys`. The current `CompositePart` id is used in the look-up in the shared or private maps to return the `BaIdList`. The `BaIdList`, with its list of `BaseAssembly` id's are then iterated through. Each id is used to look-up the corresponding `BaseAssembly` object from the persistence store. These `BaseAssembly` objects are then added to the `CompositePart`'s shared and private collections (set names in the OO7 model: `usedInPrivate` and `usedInShared`). The current `CompositePart` is also added to the current `BaseAssemblys` shared and private collections (set names: `componentsPrivate` and `componentsShared`). Thus a bidirectional link is established between `BaseAssembly` and `CompositeParts`. These collections and associations are indicated in Figure 3.1.

Figure 3.1 shows the OO7 model graph as well as the *private* and *shared* relationships. The private and shared relationships are indicated by the *ComponentsPrivate* and *ComponentsShared* diamond figures in the graph. The relationships are implemented using collections. The M and N labels represent a M:N (Many-to-Many) relationship.

3.2.1 Early implementation issues

As stated in the first phase, an `LinkingMap` class with two collections was used, These collections were stored in memory. Another possibility that was investigated, was to store this class with collections, using the persistence mechanism, in the database as well for later look-up. This approach was not used in the end because it differed from the approach used in the original OO7 C++ implementations.

In the initial part of this project, when the Java OO7 version was created, the actual `BaseAssembly` objects where stored in the `BaIdList` instead of the *id's*. This caused problems. If each phase is executed in its own transaction and that first phase transaction or session would end, the map in memory, after the first phase, would contain *detached* objects [76, 68]. "A detached entity instance is an instance with a persistent identity that is not (or no longer) associated with a persistence context" [68]. Hibernate defines an detached object as a object *which was* persistent, and which is being modified outside of a session. By looking at Hibernate's and the EJB3 definition of an detached object it is important to note that the detached object *was* in a persistence state.

A problem occurred during the linking process of phase two when the detached objects are accessed and they change to a *dirty* state and need to be persisted. Because they were detached objects some persistence mechanisms like Gemstone Facets and Versant, kept rolling back the object's values to the previous committed values. In essence the new transaction actually re-reads the object's original *persisted* values when committing, and ignored all changes made to a detached object.

A solution to this problem was to only store the id's in the list, as in the original C++ implementations of OO7, then look-up these objects during the second phase and linking process. Using this method, clean objects would be obtained at the start of a new transaction [98]. JDO calls this the persistent-clean [93] state.

Hibernate and EJB3 allows detached objects with changes to be *merged*, and to save the changes back in the persistent store. Versant and Gemstone did not allow this.

The creation of the two maps (private and shared) in memory, with their `BaIdList` collections, is not really necessary. Carey et al. state in a comment in `GenDB.c` class file, that if the `CompositeParts` are created first and then the `BaseAssemblys` there would be no need to track the id's in the collections. Carey et al. further state that they used the collection approach because this approach worked better for creating the large database configuration.

In the large configuration case (but not in the small or medium configuration cases) creating first the `CompositeParts` and then the `BaseAssemblys` caused the program to "seek all over the disk". It is unknown if this seeking is a still a issue with today's persistence stores but it was decided to use the original collection approach. Ozone [4] have used the approach of creating `CompositePart`'s first. Ibrahim and Cook [77, 78] first create the `BaseAssembly` objects and then iterate through them, creating `CompositePart`'s and establishing the links at that point.

OO7 does not specify how these linkages must be done so it was felt that these approaches need to be documented.

3.2.2 The use of transactions and indexes

Transactions are used when creating the OO7 database and when running the benchmark operations. OO7 provides two ways of using transactions to group operations. All the operations can be grouped into *one* transaction or they can be grouped into *many* smaller transactions. When the database is being created, a session is opened and transactions are committed at certain checkpoints. The following transaction strategies are used when *creating the database*:

- A transaction can be started right at the beginning of the database creation process and committed right at the end, when the whole database model has been created. Thus only one transaction is used.
- There are two *while loops* in the creation process: a while loop to create the models and its associated objects in phase 1 as discussed in Section 3.2; and a while loop to create the `CompositeParts` and its associated objects in phase 2. These two while loops can be grouped into one transaction, or a transaction per loop, or a transaction per loop iteration.

These strategies and some variations has been documented and classified during this study and are shown in Figure 3.3 and Figure 3.4. The figures shows five strategies A, B, C, D and E.

The transaction *per loop iteration* is used in our study, committing at the end of each loop iteration (strategy B). Slight differences in implementations have been found in the original OO7 C++ code. In the original Versant implementation, strategy E is used, while strategy A is used in the Objectivity implementation. It seems, though, that the Objectivity implementation did use strategy E (calling commit right at the end after 500 objects have been created instead of for every 100 objects) but the transaction code has been commented out at some point during its life time.

If creation times are reported when running the OO7 benchmark it is important to understand which strategy was used, as the strategy could influence the creation times. For example one persistence mechanisms or persistent store might be faster when the whole OO7 database is created in one transaction while another might be faster if the OO7 database is create in stages using a new transaction per stage. Our study includes the creation times so the strategy used is documented.

Transactions are also used when *running* the benchmark operations. Each operation is repeated according to the “repeatCount” setting in OO7. Five is the default repeat count which was used for all the tests. There is also an option in OO7 to run all the repeated runs either in *one transaction* or by using *many transactions.* If many transactions are used, each repeat of the operation will commit and restart a new transaction. For this study *many transactions* were used when running the benchmark operations. The study will provide details on the difference in performance in using one versus many transactions in Chapter 5.

It is important to use the same strategy for all the persistence mechanism implementations created and compared to each other.

Indexes are also not defined and created at the same time for the various implementations. Some of the implementations define indexes when the model is being defined and the index is created at the point when the object is being persisted while others define the index after the model has been persisted and thus the indexes are created right at the end. This last approach makes it very easy to add new indexes. More details on index creation will be provided in Chapter 4

A

```
createDatabase() {
start transaction

db is created
store/persist/save

end transaction/commit

}
```

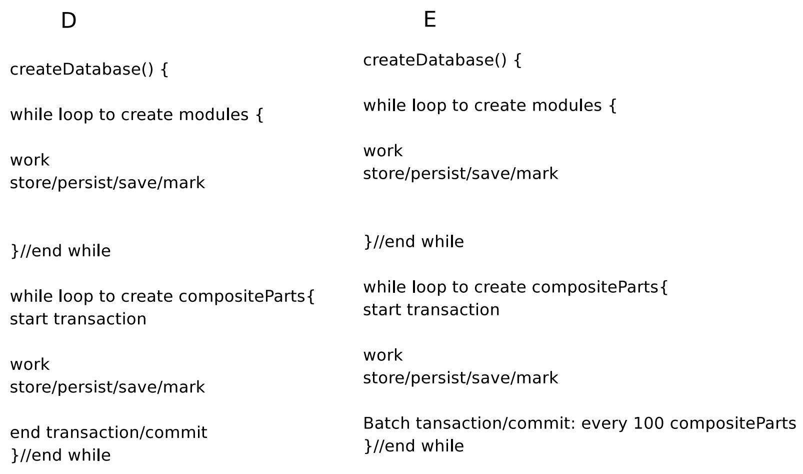
B

```
createDatabase() {

while loop to create modules {
start transaction

work
store/persist/save

end transaction/commit
}//end while

while loop to create compositeParts{
start transaction

work
store/persist/save

end transaction/commit
}//end while
```

C

```
createDatabase() {

start transaction

while loop to create modules {

work
store/persist/save

}//end while
end transaction/commit

start transaction

while loop to create compositeParts{

work
store/persist/save


}//end while

end transaction/commit
```

Figure 3.3: Grouping points and strategies A, B and C of transaction placement in OO7



D

```
createDatabase() {

while loop to create modules {

work
store/persist/save/mark


}//end while

while loop to create compositeParts{
start transaction

work
store/persist/save/mark

end transaction/commit
}//end while
```

E

```
createDatabase() {

while loop to create modules {

work
store/persist/save/mark


}//end while

while loop to create compositeParts{
start transaction

work
store/persist/save/mark

Batch tansaction/commit: every 100 compositeParts
}//end while
```

Figure 3.4: Grouping points and strategies D and E of transaction placement in OO7

3.3 Random number generation issue in OO7

The *random number generator* in Java differs slightly from the one in C++. The random number generator is used to randomly assign objects to each other and for random date creation for the build dates of objects. The date assignment is working fine but the main issue is with the randomly assignment of objects to each other.

In the OO7 model there are 729 `BaseAssembly` objects that each have 3 connections to `CompositeParts`. This means that there are $729 \times 3 = 2187$ possible connections in total. But if Traversal 1 is run, using the new OO7 Java implementation, only ± 43 680 `AtomicParts` are traversed whereas in the original OO7 C++ implementation, 43 740 objects are traversed.

To find a reason for this difference it is important to understand what Traversal 1 is doing:

> "Traverse the assembly hierarchy. As each base assembly is visited, visit each of its referenced unshared composite parts. As each composite part is visited, perform a depth first search on its graph of atomic parts. Return a count of the number of atomic parts visited when done"

To obtain a count of 43 740 `AtomicParts` traversed, the 2187 `CompositeParts` are multiplied with the number of `AtomicParts` per `CompositePart`, `NumAtomicPerComp` [48] which is 20 in the *small* OO7 configuration, and thus

2187×20 = 43 740. This means that in our Java OO7 implementation only 2184 (2184×20 = 43 680) `CompositeParts` are used rather than 2187. The same happens in Traversal 2a:

> "Repeat Traversal #1, but update objects during the traversal. There are three types of update patterns in this traversal. In each, a single update to an atomic part consists of swapping its (z, y) attributes The three types of updates are: (a) Update one atomic part per composite part. (b) Update every atomic part as it is encountered. (c) Update each atomic part in a composite part four times. When done, return the number of update operations that were actually performed."

For Traversal 2a there exists a 1-1 relationship (one `AtomicPart` per `CompositePart`) with the number of `CompositeParts` being traversed per Assembly. For the OO7 C++ implementation a count of 2187 is obtained as expected but only a count of 2184 for the Java OO7 implementation. The counts obtained for the OO7 Java implementation runs are also inconsistent, ranging from: 43 620, 43 640, 43 660, 43 680, 43 700 for Traversal 1 and 2181, 2182, 2183, 2184, 2185 for Traversal 2a.

The cause of why this number is *lower* and *different* for each run in the Java version seems to be because of the random number generator and its setup. When using a random number generator it is important that the numbers obtained are uniformly distributed. Our experiments observed that the C++ random number generator produces slightly less duplicates than the Java random generator. In the original OO7 C++ implementation the next `CompositePart` id, is obtained by the following code from the `Assembly.c` class:

compId = (o_4b)(random() % TotalCompParts) + 1;

The OO7 Java implementation has modified this code as follows:

long compId = (RandomUtil.nextInt() % SettingsUtil.TotalCompParts) + 1; in *the* `BaseAssembly` *class.*

The `RandomUtil` uses the Singleton pattern to instantiate one `Random` class object that will be used to obtain the next positive integer value by calling `nextInt()`. According to the Java API, the `Random` class "generates a stream of *pseudo-random numbers*". There are a few possibilities of how to instantiate and setup this `Random` class:

- One can instantiate a new random number generator using the default constructor `Random()`. This means that "This constructor sets the seed of the random number generator to a value very likely to be distinct from any other invocation of this constructor" according to the Java API.

- Setting a seed for using `Random(long seed)` or `setSeed(long seed)`. The Java API states that "If two instances of Random are created with the same seed, and the same sequence of method calls is made for each, they will generate and return identical sequences of numbers."

To obtain values in the required range one can either:

- use mod:
  `(Random.nextInt() % SettingsUtil.TotalCompParts)`

- or pass in the value to use as a range:
  `Random.nextInt(SettingsUtil.TotalCompParts)`

According to [103] using *mod(%)* is not recommended as "you effectively reduce the randomness of the results. The low order bits of random numbers can repeat more regularly than the entire number. This is a known issue with pseudo-random number generators, and so it's another reason not to use mod (%)." See [103] for a in-depth discussion on this topic.

Both of the setup methods for the random number generator, as well as the two methods for setting up the ranges have been used and the counts are still not the same. The Java random number generator generates more duplicates than the C++ random number generator. The reasons why this is so, is beyond the scope of the study.

All this implies that ids are generated for `CompositeParts` that already exist, which means that the Java implementation's *associations* have more *duplicate links/connections*. The Java implementation only obtain between ± 2181 and ± 2185 unique links whereas the C++ implementation seems to obtain 2187 unique links every time, or at least very consistently.

A *consistent* random generation was obtained in the end by using and setting the seed: `setSeed(1L)`. This meant that the Java implementation is able to *consistently* obtain 2185 unique connections for all the implementations. While this is slightly less than the C++ version, it is at least consistent over all the configurations and OO7 Java implementations for Hibernate, Versant and db4o.

Table 3.1 shows the *number of objects* traversed and queried in the OO7 C++ implementation versus that of the new OO7 Java implementation. The Appendix contains a table with all the descriptions of the OO7 operations that have been used in this study.

OO7 benchmark operation	Number of objects being accessed in the original C++ implementation	Number of objects being accessed in the new Java implementation
Trav1	43 740	43 700
Trav2a	2187	2185
Trav2b	43 740	43 700
Trav2c	174 960	174 800
Trav3a	2187	2185
Trav3b	43 740	43 700
Trav3c	174 960	174800
Trav6	2187	2185
Trav8	2632 of character in manual	1
Trav9	0	0
Query1	10	10
Query2 (Range)	93 (Date range 1%)	100
Query3 (Range)	965(Date range 10%)	970
Query4 (Random selection)	45 (document, baseAssembly pairs)	42
Query5(Randomness: uses baseAssembly. getComponentsPrivate() that are generated and randomly linked)	233	199
Query7	10 000	10 000
Query8	10 000	10 000
Insert and Delete	10 compositeParts 200 atomic parts	10 CompositePart's 200 atomic parts

Table 3.1: The number of objects being traversed and queried in the small database configuration with 3 connections

3.4 Java version of OO7 Benchmark

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

3.4.1 Overall approach

The Java version was written in two stages. An initial attempt took the lead from another Java OO7 version that had been developed and distributed by the Ozone open source object database [4]. On investigation, it became apparent that Ozone's Java version was very basic. Not only was it not a full equivalent of the original C++ version, but valuable debugging and configuration settings available in the C++ version had not been incorporated into the Ozone version.

In a second stage, the original C++ code that had been used for the Versant implementation was taken as a starting point. An equivalent Java class was written for each C++ class. Here all of the language structures that were similar were copied over to the Java version. An example of this was where similar language structures (such as for-loops) were merely copied over from the C++ to the Java version. This was done in an effort to stay as close to the original version as possible. This easily led to a Java code equivalent of OO7, but which did not yet provide any persistence-related code. However, this was useful in and of itself, as it was deemed desirable to have a pure in-memory representation of the model. This was to serve as an absolute baseline in comparing cold and hot object operations across different platforms. The idea for doing this was taken from Jordan [80]. Once the pure Java OO7 model was in place, the db4o, Versant and Hibernate implementations were produced. These will be discussed in Chapter 4.

3.4.2 Utility classes

A `Persistence` class was designed for use in all the persistence mechanisms tested (db4o, Hibernate and Versant). This class was used at all points in the benchmark model where persistence was needed. It has methods for saving, deleting, updating, etc. hiding the specific implementation details.

Each implementation then implemented its persistence code in a separate utility class whose methods were called by methods in the `Persistence` class. Followers of Design Patterns may recognise this as an example of using the Bridge pattern. The db4o utility class was called `Db4oUtil`, Hibernate's utility class, `HibernateUtil` and Versant uses the `VersantUtil`. The intention was to enable us to hide the persistence mechanism used from the model. The idea was to use only one model, and to merely interchange the correct persistence mechanism classes when testing the different platforms. This aspiration was not realised, as it was found that Hibernate required that the owner of a hierarchical object tree or associated objects had to be saved before saving objects on the lower levels. This was not true for db4o and Versant. Thus, the order of calls to the persistence utility had to be different in the implementations. Also the way indexes are created differed for each persistence mechanism. This necessitated separate implementations for the platforms. The original C++ OO7 implementation also used this approach of writing separate implementations: one for Versant, one for Objectivity, etc.

Jordan [80] reports that he

> "abstracted the interface to the persistence mechanism through an interface called Store. This interface provides methods to set/get the root object, begin/end a transaction, and other miscellaneous methods needed to interface with the persistence mechanism."

Jordan [80] also reports that he created a PMStore for each persistence mechanism. The `Persistence` class in our study used a similar approach. The implementations would call methods on the `Persistence` class which in turn would call the correct utility class for the current persistence mechanism. Currently these utility classes are implemented using *static methods*. This means that there is only one handle to the underlying persistent store. On reflection this was *not a good design choice* and will be changed in future.

In addition to these, it is also possible to call query utility classes to test different query mechanisms. An example of these are in the db4o implementation: `Db4oNativeQueries` and `Db4oSodaQueries`. Separate query classes are valuable because the actual query code is then moved out of the persistence mechanism utilities like `Db4oUtil` and into specific query utilities.

Our work also uses the OO7 queries operations, unlike Jordan [80] where they only tested OO7 traversals.

3.4.3 Deviations from the original benchmark

The formula for computing the average hot times was changed by including the time taken for the last run. Carey et al. [46] had omitted the last run because they didn't want to include the overhead of commits. Our study argues that commits are an inherent part of the operation and they may have quite an important influence on the times. Thus, the *average hot time* calculation has been changed to include all of the iteration times except for the first, this iteration time being for the *cold run*.

Another deviation from the original benchmark relates to documentation text associated with a `Document` object. The text is used by a search to find certain characters, a pattern matching test of *large text objects*. While a basic text string is included in our `Document` objects it was felt that searching through a *large texts* and *large objects* was not essential to our study. Thus, in this study, the time measurements for Traversals 8 and 9 will be less than they would have been if study had not deviated from the original OO7 benchmark.

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

3.4.4 Run scripts

To create and run the OO7 benchmark, Ant [7] scripts were used as the build and run tool. This meant that for each of the configurations for the persistence mechanism and persistent store, an Ant *script was created* to run the benchmark operations. Figure 3.5 shows how the Ant build files are being used. The example shows the build files for Hibernate.

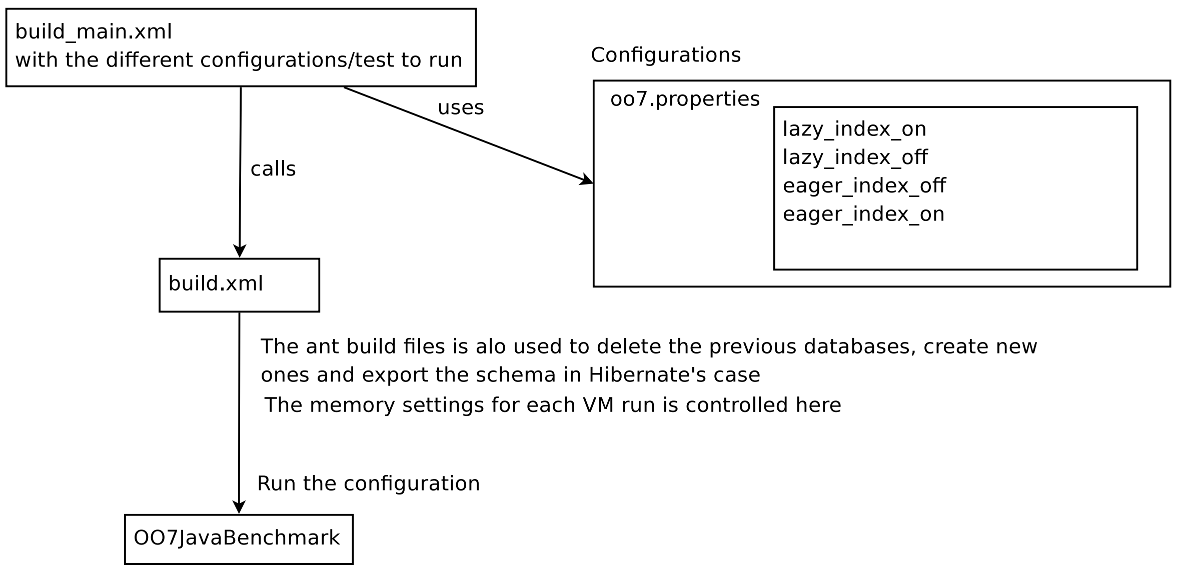

build_main.xml
with the different configurations/test to run
uses
calls
Configurations
oo7.properties
lazy_index_on
lazy_index_off
eager_index_off
eager_index_on
build.xml
The ant build files is alo used to delete the previous databases, create new ones and export the schema in Hibernate's case
The memory settings for each VM run is controlled here
Run the configuration
OO7JavaBenchmark

Figure 3.5: Ant build process for OO7 Java benchmark

Figure 3.6 below shows a snippet of the original OO7 benchmark commands taken from the `Run.one` script.

```
stopdb ${db}; startdb ${db}
echo "Starting: ./bench oo7.config.${model} 1 t2a one" > ${TERMINAL_PATH}
./bench oo7.config.${model} 1 t2a one
stopdb ${db}; startdb ${db}
```

Figure 3.6: Snippet of original run script

The script shows that the database is started and stopped for *each operation* of the OO7 benchmark. It was found that some Java implementations started the database and then ran *all the operations* before closing the database [77]. It was also found that others tried to modify our implementation using the approach shown in Figure 3.7. This approach would create the database then immediately run the operations. Using the approach shown in Figure 3.7 and the approach of Ibrahim and Cook [77], a situation would be created where the cache is already filled with objects after the database is created. The cache is also filled after the first run of an operation. It would then be difficult to ascertain how the *cache* influences a *specific operation* because the same cache is being used for all operations.

Start VM and Start database

Create database

Run Traversal 1 x times

Run Traversal 2a x times

Run all operations……
………………………………

Stop VM and Stop database

Figure 3.7: Running the OO7 Java benchmark in one process and VM

Carey et al. [48] state that a lot of effort was spent on clearing all the caches between runs. It is important to clear the cache between operation runs to clearly see the *effect of caching* when a *specific operation* is run. For this reason our study's Ant scripts followed a approach similar to the original scripts. The Ant scripts first starts-up the benchmark application in a new VM and also starts the database. Then the operation is run `x` number of times. After the operation has been run the database is closed and the application, as well as the VM, are stopped. *For each operation* the Ant script starts up the benchmark application in a new VM and also starts the database. This approach is illustrated in Figure 3.8.

Start VM and Start database

Create database

Stop VM and Stop database

Start VM and Start database

Run Traversal 1 x times

Stop VM and Stop database

....

Figure 3.8: Running each operation of the OO7 Java benchmark in a new process and VM

The next chapter will discuss the specific Java implementations for the persistence mechanisms and persistent stores that were benchmarked, as well as the their benchmark configurations.

Chapter 4

OO7 Java implementations

The details of the Java OO7 implementations that were created for the object databases and ORM tool being tested (db4o, Hibernate and Versant) are provided in this chapter.

As mentioned in Chapter 3, both the db4o, Hibernate and Versant implementations to persist objects were derived from the Java OO7 in-memory code version that simply generated and stored objects in memory.

Often, adding persistence to a Java application requires rather intrusive additions and changes to the code. There are really three categories of changes required: telling the object database or ORM Tool *what* to persist, *how* to persist, and *when* to persist.

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

The following sections discuss how the Java implementation were created for db4o, Hibernate and Versant. It also discusses how objects were marked to be persistent for each implementation.

4.1 db4o implementation

db4o[61] is an open source object oriented database that easily and without extra work to tell it what and how to persist, stores Plain Old Java Objects (also known as POJOs). db4o does however require one to tell it when to persist. In first versions of the implementation, code to save or update objects in the persistence store was inserted just after the object had been created or changed in memory. This approach has been changed to use the *one* or *many* transaction strategies as described in Section 3.2.2. Strategy B was used for creating the database and *many transactions* was used when running the operations of the benchmark. At the commit points the new objects are first saved (persisted) using the `ObjectContainer.set(object)` method of db4o, then the current transaction is committed using `ObjectContainer.commit()`.

By adding transactions and the persistent calls, producing the db4o implementation from the OO7 Java code was in fact quite straightforward.

When using db4o, one needs to set the *update depth* and *activation depth* parameters [61, 62]. These settings are used to control performance.

The default update depth in db4o is 1. If the default update depth is used and `ObjectContainer.set(objectX)` *is* called then only *objectX's* changes are persisted. If the update depth is set to a value greater than 1, then `ObjectContainer.set(objectX)` updates not only *objectX* but also objects referenced by *objectX*. Note that if an object is saved the first time its associated objects are also saved.

Since the benchmark makes use of objects that have sets of references to other objects, and since changes should be saved immediately, this default parameter was changed. The update depth can be set to cascade when an object is saved or updated. It was found that the *update depth* of only these objects: `BaseAssembly`, `CompositePart` and `Module` needed to be changed to `cascadeOnUpdate(true)` to store all the changes to the model. The rest of the object's update depth was kept at the default value.

Activation depth relates to retrieving objects from the database that reference other objects. db4o can, if needed, retrieve the whole object graph referenced by the object being retrieved at once. This is called *eager loading*. This can be inefficient, especially if there is no immediate need to use the entire object tree right away. So, when an object is retrieved that has references to other objects, retrieving of the referenced objects can be deferred until needed. This is a technique called *lazy loading*, where referenced objects are fetched only when a reference to them is followed. The tree depth at which this occurs is called activation depth in db4o. By default, db4o specifies activation depth as 5. The activation settings used for db4o will be discussed in the following section on configurations.

4.1.1 Configurations

As stated in Chapter 1 one of the goals of the study, is to investigate the impact of different performance techniques on the performance of the object databases and ORM tools. For this reason the following test configurations of db4o were run:

- Running db4o with *eager loading* on and *indexes off*. This configuration tests the worst performance of db4o with no advanced performance configurations settings on. In running the benchmark, the activation depth was increased to retrieve the *whole* referenced object graph when eager loading was being tested. This meant that `Db4o.configure().activationDepth(Integer.MAX_VALUE)` was set to the maximum for all the OO7 model classes. Indexes can be turned on or off by using the created boolean property called *useIndexes.* db4o defines indexes on class properties/fields using `Db4o.configure().objectClass(ClassNameX.class)` `.objectField("fieldY").indexed(true).`

- Running db4o with *eager loading* on and *indexes on.* This test is used to investigate if the use of indexes can improve the performance even if the whole object graph is being retrieved into memory.

- Running db4o with *lazy loading* on and *indexes off.* There are currently two approaches to enable lazy loading in db4o. Approach *one* is to use the newly created *transparent activation framework [62]* available in version 7 of db4o. Approach *two* is to manually activate the objects when accessed. At the time of comparing db4o for this study there was no stable version of db4o 7 and the *transparent activation framework* available. This meant that approach *two* was selected as it is the older and more stable approach. If *transparent activation* had been used it would have detected automatically which objects needed to be retrieved into memory at the correct points and do so automatically. Transparent activation works by implementing an interface called Activatable [62]. Manual activation meant that objects were activated by calling `ObjectContainer.activate(object, depth)` and by calling `ObjectSet.next()`when using queries. The default activation depth in db4o was set to 1. To be able to switch between eager loading and lazy loading, a boolean property `USE_LAZY_LOADING` was added. The `Traversal` class and some of the query (Query 4 and 5) classes needed to be modified to include the new activation code. For lazy traversals a new class called `Db4oLazyTraversal` was added. `Db4oLazyTraversal` class is similar to the `Traversal` class but includes code to activate objects. This activation is needed, otherwise with an activation depth of 1, many null pointer exceptions would be thrown when objects are not retrieved or activated in memory. Another approach, that was not followed in the end, was to modify the original *Traversal* class by adding a check for `USE_LAZY_LOADING` all over the class. In the end it was decided to create a separate class (`Db4oLazyTraversal`) and then add an option to choose between lazy and eager in the main class (`OO7JavaBenchmark`). The main class creates and runs the benchmark, and switches between using the normal *eager* loading classes or the *lazy* loading classes. Query 4 and 5 were modified because, while most queries activated the needed objects by calling `ObjectSet.next()` query 4 and 5, while iterating through the object result sets, retrieved other attached objects that needed to be manually activated. All the other operation classes worked fine with activation depth 1 and the queries all used `ObjectSet.next()` to activate objects.

- Running db4o with *lazy loading* on and *indexes on*. This test is used to investigate the impact of using indexes and lazy loading.

- Running db4o with *lazy loading* on and *indexes on* in a embedded database or standalone mode. Embedded database or standalone mode will be discussed further in the next section. The standalone mode was added because it is the default mode for db4o. The study wants to find the difference in performance between the standalone and networking mode. Note that when the model is created and saved to the database in standalone mode an extra *cascade on update depth* setting was needed on the `CompositePart` class to store the whole model. The reason for this difference between the standalone mode and the networking mode has not been investigated in this study.

These configuration values are mostly set in the `DB4oUtil` and `StartServer` class, before the db4o container is opened.

4.1.2 Architecture setup

db4o can be run as an[62, 92]:

- embedded database (standalone mode),

- as a local server in the same virtual machine (an embedded *server* mode).

- or as a network server using TCP/IP for communication (networked mode).

In the early experiments [96] the *standalone mode* was used, which meant that the database file was accessed directly from the OO7 benchmark application and from inside *one VM* using no networking communication. Subsequently db4o was run in *networked mode*, using a network server (client/server mode) and using TCP/IP network communication. The server and client are then started, each in its *own VM*. Because Hibernate and Versant were using network communication between client and server it made sense to make this change and make comparisons more fair. The architecture setup is displayed in Figure 4.1.

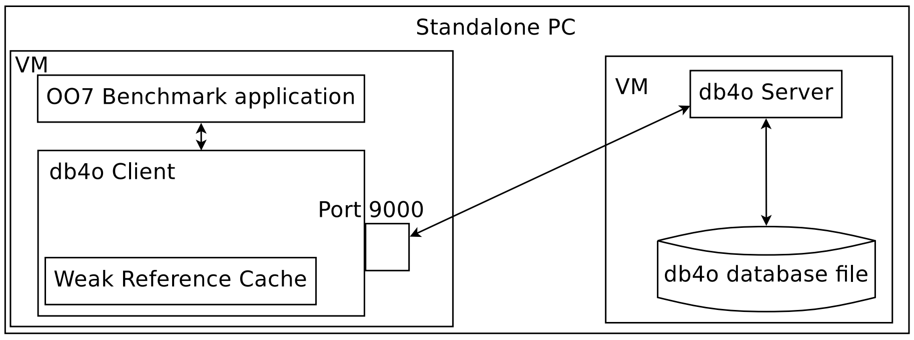


Figure 4.1: Architecture setup of db4o

db4o uses a *weak reference cache* which is associated with an open database or `ObjectContainer`. A container represents a database. In client/server mode each client will have its own reference cache [62]. It is important to note that whenever the db4o database container is opened a transaction is started, and each time a commit is called, the transaction is ended and a new transaction is immediately created. Thus there is always an active transaction.

Because each client has its own *reference cache* it can get complicated if object A is cached by many clients, as changes need to be *synchronised.* The db4o reference manual [62] states that each client must “refresh them from the server when they get updated” using committed callbacks in db4o. For our implementation and study *only one client* is used to run the operations on the OO7 model, so no synchronisation was needed.

Java provides different types of *references,* which are strong or normal references, phantom references, soft references and weak references [79]. [22] states that “An object that is not strongly or softly reachable, but is referenced by a weak reference is called weakly reachable“. If objects are added to the cache, using weak references, they can be garbage collected if they later become unreachable or there are no more references to these specific objects.

In the db4o reference manual [62] it is stated that if weak references are turned off the speed of the system will improve but that memory usage will increase. For this study the default weak reference cache was used.

It is interesting to note that db4o does not support standards like JDO and JPA directly. However there are currently implementations of JDO and JPA available for use with db4o from JPOX [9].

4.2 Hibernate implementation

Hibernate is an ORM tool that stores in-memory objects to, and retrieves in-memory objects from, a relational database. Hibernate maps objects from the pure object model to a relational model. To create the Hibernate OO7 implementation, the in-memory object model of the OO7 Java code needed to be modified to make use of Hibernate. Hibernate can be used with any relational database – in this study PostgreSQL [14] was used.

To more clearly understand how the in-memory and pure object model was modified to make use of Hibernate, certain concepts need to defined. Ambler [31] provides the following definitions related to mapping object models to a relational model in an relational database:

- "Mapping: The act of determining how objects and their relationships are persisted in permanent data storage, in this case a relational database.
- Property: A data attribute, either implemented as a physical attribute, such as String `firstName`, or as a virtual attribute implemented via an operation, such as Currency `getTotal()`.
- Property mapping: A mapping that describes how to persist an object's property.
- Relationship mapping: A mapping that describes how to persist a relationship between two or more objects (generally, association, aggregation, or composition).
- Inheritance mapping: Mapping the inheritance hierarchy to relational database tables."

"Hibernate like all other object/relational mapping tools, requires metadata that governs the transformation of data from one representation to the other (and vice versa)" [74]. In using Hibernate, these "mapping metadata" [74] can be specified in XML mapping files or in the Java code by using Xdoclet [6] tags or JPA (Java Persistence API) annotations [74, 68]. In the early experiments [96], using Java 1.4, Hibernate Xdoclet tags were used in the code. By using Xdoclet the tags was extracted from the Java files, using an Ant task, to create XML mapping files. Currently Java 5 is being used and Xdoclet tags have been replaced by JPA annotations. The JPA `@Entity` tag is used to mark objects as *persistent* objects. From the JPA annotations in the classes, a basic XML mapping file (.hbm) is extracted using Hibernate's Ant `hibernatetool` and `hbm2hbmxml` tasks [75]. The extracted mapping file is then duplicated for each of the test configurations that are going to be used in the benchmark tests. Each file is changed manually to include the specific Hibernate tags that will be needed for that specific test. An example of this manual change is to include lazy="false" to enable eager loading, or lazy="true" to enable lazy loading. Section 4.2.4 discusses the Hibernate test configurations.

In the OO7 model there are properties, relationships and inheritance which all need to be mapped using Hibernate. The mapping of the properties was relatively straightforward. Most of the time was spent on getting the relationship and inheritance mappings to work correctly. These mappings are discussed in the following sections.

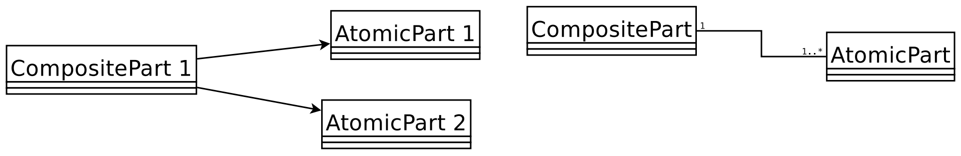
In the previous section it was mentioned that when using db4o, one can set the activation depth of the objects retrieved. It is interesting to note that Hibernate also provides lazy loading, where it is possible to limit the number of objects returned.

4.2.1 Relationship mappings

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

All the associations in the OO7 model are bi-directional, and so one has to consider how to specify relational mappings to allow queries to follow relationship navigation from both ends. The *one-to-many* and *one-to-one* relationships were implemented as ordinary relational database primary key / foreign key entity relationships using the `@OneToMany` and `@JoinColumn` JPA annotations. To elaborate: having a class A and a class B in the *object* *model* meant having TableA and TableB in the *relational model*, with each row in the tables storing the corresponding objects' attributes. If there is a one-to-many relationship between class A and class B, the relationship is represented as a column in TableB which keeps *primary keys from TableA*. These keys are *foreign keys* in TableB. To access the B objects from an A object, select all of the rows in TableB that reference the A object's primary key as a foreign key. Going the other way, to access the A object from any B object, select the row in TableA that has the stored foreign key as its primary key. One-to-one mappings were implemented using the `@OneToOne` JPA annotations.

There are two one-to-many mappings associations in the OO7 model. These are the *"subAssemblies"* and the *"parts"* associations shown in Figure 3.1. An example one-to-many mapping ("*parts*"), which uses a `@JoinColumn` tag, related to the OO7 Java Hibernate implementation is shown in Figure 4.2.



composite_parts table

composite_part_id(PK)	Property x
1	x1

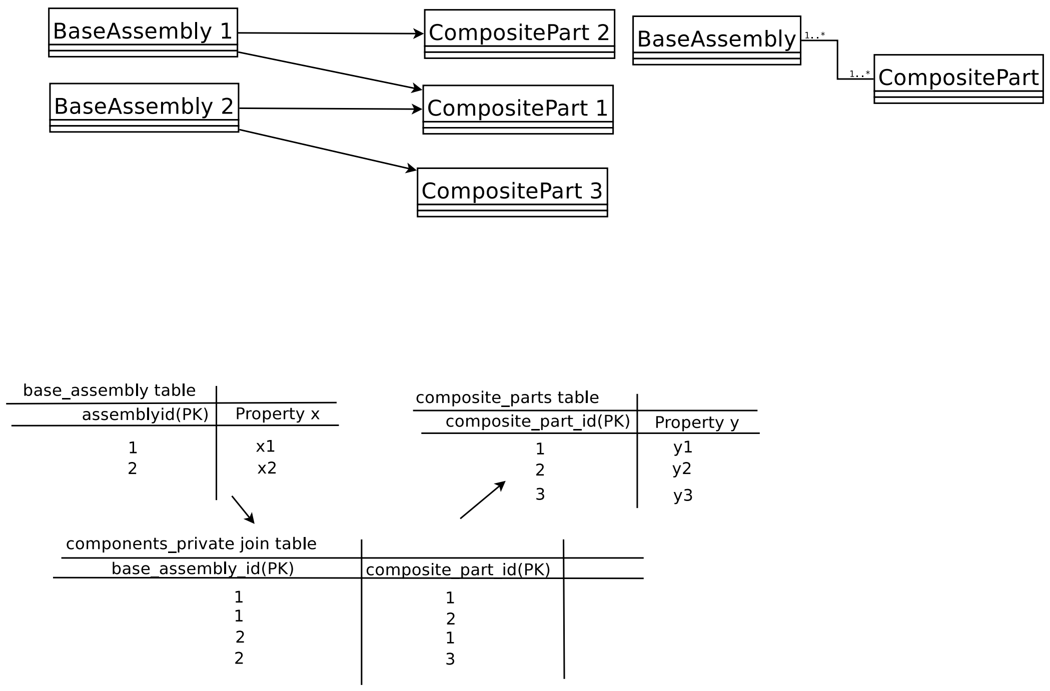
atomic_parts table

atomic_part_id(PK)	composite_part_id(FK)	Property y
1	1	y1
2	1	y2

Figure 4.2: An example showing one-to-many mapping tables

The other one-to-many mapping, *"subAssemblies"*, was tagged with `@OneToMany` and uses an join table for the mapping. For all the *many-to-many bi-directional* associations, join tables were used. A join table stores, separately from the referenced tables, the primary keys of the related entities. Extending the example above to a many-to-many relationship between A objects and B objects, this means the provision an additional table, say TableAB, with columns A and B storing the list of keys of related A's and B's. To access the related B objects from an A object would mean executing a query that found all of the related B objects' keys from TableAB, and then retrieving those rows in B that had those as primary key. Navigating the relationship from B's to A's is done similarly. For these mappings `@ManyToMany`, `@JoinTable` *and* `@JoinColumn` JPA annotations was used.

An example many-to-many mapping related to the OO7 Java Hibernate implementation is shown in Figure 4.3.



base_assembly table

assemblyid(PK)	Property x
1	x1
2	x2

composite_parts table

composite_part_id(PK)	Property y
1	y1
2	y2
3	y3

components_private join table

base_assembly_id(PK)	composite_part_id(PK)
1	1
1	2
2	1
2	3

Figure 4.3: An example showing many-to-many mapping tables

4.2.2 Inheritance mappings

According to [31, 50, 89, 39], four basic approaches can be used to map object oriented inheritance structures to a relational model:

- Map the whole class hierarchy structure to one table, i.e. one table containing all of the combined attributes. In Hibernate this is called the *table per class hierarchy (single table)* strategy
- "Map each concrete class to its own table": i.e. all of the subclasses that inherit from an abstract class get their own table. This is called the *table per concrete classes* approach in Hibernate.
- "Map each class to its own table". Hibernate calls this the *table per subclass* strategy.
- "Map the classes into a generic table structure".

The Hibernate supporting literature provides strategies to enable one to use any of the approaches mentioned above [76, ch 9]. In the OO7 Java

Hibernate code, in general, each class was mapped to its own table. This *table per subclass* strategy [76] was selected because it is a straight forward one-to-one mapping. The strategy of *mapping concrete classes* to their own tables was used only in one place, namely to map the OO7 abstract class called `DesignObject` and its subclasses: `Module`, `Assembly`, `CompositePart` and `AtomicPart`. Figure 3.1 shows `DesignObject` and the classes that inherit from it.

The mapping of *concrete classes* approach generally decreases the number of tables that are needed, compared to the *table per subclass strategy.* However, mapping of *concrete classes* approach results in duplicated information, since the properties of the parent class have to be included in each child class table [31]. If properties are added later to the parent class, then they must be inserted into each child class table.

One should keep this complexity in mind when choosing mapping strategies [31]. As the benchmark scenario did not include the addition of properties, this complexity was not an issue. Furthermore, the properties in child classes were quite simple: a build date, a string type and an id. (Note: it was necessary to rename the original 'id' field to 'design_id' as there were some conflicts in Hibernate with the name 'id'.)

Figure 3.1 shows that `ComplexAssembly` and `BaseAssembly` objects inherit from Assembly. For this inheritance mapping the *"joined" subclass strategy* [74] was used.
The JPA `@Inheritance(strategy=InheritanceType.JOINED)` annotation was used for this inheritance mapping. A table was created per child class (base_assembly and complex_assembly) and a main table for the parent class (assembly table). The main table contains all the properties of the class and the foreign key id's of the child tables. The main table also contains a column that displays the type of object the row represents which is either a `ComplexAssembly` or a `BaseAssembly` object.

Keep in mind that during this mapping process, objects and their inheritance structure can be lost if the mapping is not correctly done [89].

4.2.3 Impedance mismatch difficulties

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

It is clear that this kind of manual mapping requires fairly intimate knowledge of both the object and relational models, and the mechanics of the chosen mappings. For example, when mapping `AtomicParts` and their `Connections`, a problem was encountered that related to the order in which the objects had to be saved. This was manifested as a *referential integrity constraint* in the database. Elmasri and Navathe [69] describe a referential integrity constraint as the requirement that a tuple in one relation may not refer to a tuple in another relation unless the latter tuple already exists. This meant that an `AtomicPart` object had to be saved first before saving a `Connection` object that is linked to it.

JPA annotations greatly improved the mapping experience when using Hibernate.

4.2.4 Configurations

For the Hibernate implementation five test configurations were created. These are as follows:

- Running Hibernate with *lazy loading* on and *indexes* on using *JPA annotations*. This was the basic configuration from which modified XML mapping files (.hbm) were created for each of the other test configurations. The study is interested in using annotations and comparing their performance to that of using mapping files. The process of creating the XML mapping file from JPA annotations is discussed in Section 4.2. `@Index` annotations was used to tag fields as indexes. To enable lazy loading `fetch=FetchType.LAZY` was specified on all `@OneToOne`, `@ManyToMany`, `@OneToMany` and `@ManyToOne` annotations. The annotation configuration also uses Hibernate Named Queries [74].

- Running Hibernate with *eager loading* on and *indexes off*. This configuration tests the worst performance of Hibernate with no advanced performance configurations settings on. To enable this configuration indexes was removed by removing the index="xxx" property on index fields and setting lazy="false" on collections and associations in the XML mapping file.

- Running Hibernate with *eager loading* on and *indexes on.* This test investigates if the use of indexes can improve the performance even if the *whole object graph* is retrieved into memory. To enable this configuration indexes are added by adding the index="xxx" property on index fields and setting lazy=”false” on collections and associations in the XML mapping file.

- Running Hibernate with *lazy loading* on and *indexes off.* To enable this configuration indexes have been removed by removing the index="xxx" property on index fields and setting lazy=”true” on collections and associations in the XML mapping file.

- Running Hibernate with *lazy loading* on and *indexes on.* This test investigates the impact of using indexes and lazy loading. To enable this configuration indexes are added by adding the index="xxx" property on index fields and setting lazy=”true” on collections and associations in the XML mapping file. This specific configuration will be used to compare against the *annotation configuration.*

Ant *targets* have been created for each of these configurations. These targets include the correct Hibernate mapping files and will “export”, using the Hibernate schema export tools [75], the correct database schema. By exporting the schema, the database is created from scratch, with or without index fields according to the mapping files used. If an earlier database existed, using a different configuration, it will be deleted.

4.2.5 Architecture setup

Hibernate uses JDBC (Java Data Base Connectivity) to connect to a relational database which in this study is PostgreSQL. The PostgreSQL server is started up first and then connections are made by Hibernate through the JDBC API. JDBC is an API that provides access to relational databases (data sources) from Java and wraps SQL statements and results “in an object layer” [40]. Hibernates creates a `Session` object for each unit of work that it is currently busy with. The session is created by calling the `SessionFactory`. The session factory represents the data source. Creating a `SessionFactory` is expensive so it is usually created once per application or per database being accessed, and from there sessions are obtained for the unit of work requested [39].

The session will also create a JDBC connection to the database when needed. When Hibernate opens a session, the session has an associated cache which is “a transaction-level cache of persistent data” [76]. This cache is also called the *first level cache*. In Bauer and King [39] a session is described as "a cache or collection of loaded objects relating to a single unit of work". An extra *second level cache* can be configured on classes or collections. *In the current study second level caches were not used unless stated otherwise.*

Hibernate provides transactions through the use of JTA (Java Transaction API) or JDBC transactions. The Hibernate Transaction API was configured to use the standard JDBC transactions. Whenever a session is started a transaction is also started and associated with the session. When calling commit the transaction is committed. The session and the session factory is only closed right at the end and is not included in the timings of the benchmark operations. All this implies that one session and many transactions are used per test configuration. *It is important to take note* of the *"session-per-operation" anti-pattern* as defined in the Hibernate manual [76]. The anti-pattern states that it is not good practise to open and close a Session for each interaction with the database in one thread. The current study avoids using "session-per-operation" in all cases. It is also important to note that *how* and *when* a session and transaction is used, could influence benchmark results and it is important to document how they have been used.

Figure 4.4 shows the architecture setup used for the Hibernate benchmark tests. *It is important to note* that PostgreSQL is running separately from the benchmark and Hibernate.

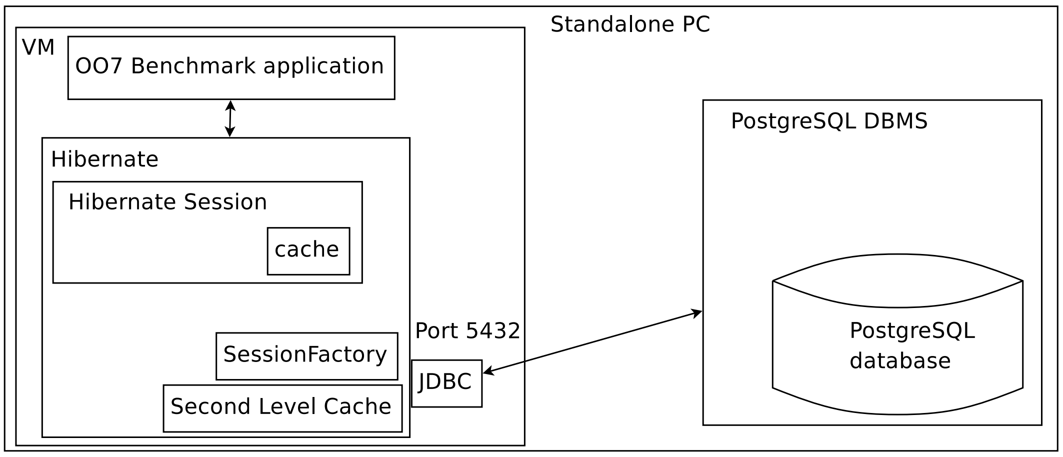


Figure 4.4: Architecture setup of Hibernate

4.3 Versant implementation

Versant [15] is a commercial proprietary object oriented database for Java and C++. Versant was not included in the original OO7 benchmark study undertaken by Carey et al. [48] but was included in a later study by them [46]. For this study the Versant OO7 implementation was created by adding the Versant persistence code to the basic *in-memory* implementation. This is the same approach which was used for the other implementations.

Versant provides the following API's to persist Java objects:

- Java Versant Interface (JVI), which consists out of the *fundamental* JVI, and the *transparent* JVI;
- ODMG JVI which is based on ODMG 2.0 Java Bindings [98]; and
- Versant JDO Interface (Java Data Objects) implementation for Java [99, 98].

It was decided to use the JVI *transparent* interface of Versant. This decision was based on the fact that neither ODMG nor JDO were used in the Hibernate and db4o implementations as neither of these support ODMG and JDO.

Looking closer at Java Versant Interface, the transparent JVI is a layer on top of the fundamental JVI [98]. It is possible to use the wrapper API's of the transparent JVI, which is a subset, or directly use the fundamental JVI API's. The JVI "automatically maps the classes and fields of persistent Java objects to the Versant object model". The transparent JVI along with the underlying fundamental JVI is used to persist objects, to create sessions with associated transactions and caches, to provide for error handling and to implement queries.

Versant uses a *post-processor byte code enhancer*, called the enhancer tool, and a configuration file to enhance the Java class files. The enhancer can be run as a standalone utility or as a Java class loader. The standalone utility has been used for this Versant implementation as the class loader, while being more transparent and dynamic, includes an overhead as the classes are enhanced at run-time. Versant specifies certain persistence categories of which *persistent capable* and *persistent aware* were used for this Versant implementation. These categories are specified for each class in the configuration file used by the enhancer. The enhancer is used for example [98]:

- to make Java objects persistent capable and persistent aware;
- from the code, to generate the database schema;
- to include code to know when an object has been changed so that it could be marked as dirty; and
- to add transaction and locking (read and write locks) related code.

Persistent capable classes *can be stored* in the database and persistent aware classes are classes that operate on persistent capable classes. Persistent capable classes can either be in a persistent or transient state.

The enhancer also provides code to read objects into memory from the database *once they are accessed*. This, as stated in earlier sections, provides *lazy loading* of accessed objects. To make transient, persistent capable objects, persistent in the database the `makePersistent(object)` method of the transparent JVI must be used. Versant uses *transitive persistence* or *persistence by reachability* to make objects and their associated object reference tree persistent. This is similar to the *update depth* of db4o.

Versant provides support for the normal Java collections, as second class objects [98], and arrays as well as enhanced *persistent collections* (`VVector, DVector, LargeVector` and `VHashtable`) and the ODMG standard collections (`ListOfObject, SetOfObject, BagOfObject` and `MapOfObject`). This study tried to use normal Java collections for all implementations, but a problem was experienced in the Versant implementation for the *shared* and *private* collections in the `BaseAssembly` and `CompositePart`. When the collections were modified in the two model creation phases, changes to these collections were not saved. According to the Versant Forum "Starting VOD 7.0.1.4 all supported collections in java.util support a concept called automatic change-tracking". This feature should assist when SCO's (second class objects) are changed. This study had been using `VVector` and VOD (Versant Object Database) 7.0.1.3 for a while. Versant was upgraded to VOD 7.0.1.4, to test the use of normal Java collections. However, none of the changes to the collections were persisted. As a consequence only the *shared* and *private* Java collections were replaced with Versant `VVector` collections.

The rest of the collections use normal Java collections.

The normal Java collections (java.util package) are stored as second class objects (SCO) in the Versant database. 'The Second Class Object is in some sense "subordinate" to a First Class (sic)' [98]. These SCO's are serialised by the Java Serialisation API. "This serialization can result in a performance loss with large objects that contain references to many persistent objects" [98, page 198]. Looking at the results for Versant in the next few chapters there does not seem be a performance loss when using normal Java collections. The results for the Versant implementation is discussed in Chapters 5 and 6.

The OO7 code is written to operate on Java collections. In db4o when queries are executed, which involve a collection being returned, an `ObjectSet` is returned which is cast to a Java List object. Hibernate queries return Java List objects automatically.

In Versant, collection results, using Queries 6 API (VQL 6.0) [98], are stored and returned as a `VEnumeration`. This caused casting problems because the rest of the Java OO7 benchmark code was expecting Java collections. To solve this problem all `VEnumeration` results where iterated through, and added to a Java collection. This newly created collection was returned for use by the Java OO7 benchmark.

Queries 7.0 is the new Query API that became available during the study and will be used in future studies.

4.3.1 Configurations

For the Versant implementation there are less test configurations than the others. Versant does not provide features to enable lazy or eager loading. When Versant injects persistence code into the classes, Versant seems to enable lazy loading automatically. Versant's *persistence collections* also enable lazy loading when objects are accessed. This meant that there was no way to enable eager loading. The Versant test configurations are:

- Running Versant with *lazy loading* on and *indexes* and using `commit()`.
- Running Versant with *lazy loading* on and *indexes* off and using `commit()`.

The following two extra tests were also included to investigate the impact of not clearing the cache:

- Running Versant with *lazy loading* on and *indexes* on and using `checkpointCommit()`.
- Running Versant with *lazy loading* on and *indexes* off and using `checkpointCommit()`.

The cache setup and the difference between `commit()` and `checkpointCommit()` will be discussed in the next section.

4.3.2 Architecture setup

The Versant database daemon is started first on operating system startup. Then using Versant commands, the actual databases can be created, started and stopped. Java clients are now able to connect to a database using sessions. Each session is associated with one database. When a session is created, a transaction is created and associated with that session. There is a one-to-one association between a session and a transaction [98]. When a session is ended the transaction is committed as well. Each session has an associated "object cache of persistent objects" [98]. The cache has the following properties [98]:

- "Occupies memory and contributes to the process size.
- Consumes none of the Java heap."

The cache that is associated with the session is created in the "C memory space to speed up access to objects in a transaction" [98]. The cache contains objects associated with the current running transaction. When a object is needed the cache is searched first and if missing, retrieved from the database and placed in the session cache. The session can be committed using either `commit()` or `checkpointCommit()`. While both end the current transaction and start a new transaction the major difference is that a `commit()` clears the object cache while the `checkpointCommit()` does not clear the cache.

The cache also has an COD (Cached Object Descriptor) table. The COD table contains address pointers to the objects on disk and will generally make retrieval of these objects faster [100]. The COD table is not cleared when the transaction ends and only gets cleared when the session is ended. *As with Hibernate and db4o one session and many transactions, per configuration* is used.

The *standard* Versant session was used during this study.

Versant provides B-tree and hash table indexes which can either be *unique* or *normal* [98, 100]. For the Versant OO7 implementation, unique type indexes were used. A unique B-tree index was used for the `AtomicPart.buildDate` property, while the rest used hash indexes. The index setup is the same as in the original Versant C++ implementation.

Versant provides many configuration settings. These settings include setting how the server buffer will be flushed, locking and logging settings, the database volume sizes, extra tuning parameters, etc. For this study the default settings were used. It is important to know whether logging and locks are enabled as these could have a impact on the performance of the persistence mechanism being tested. For all databases and ORM tools in this study transactions (with locks enabled), as well as logging were used.

Figure 4.5 shows the Versant architecture diagram.

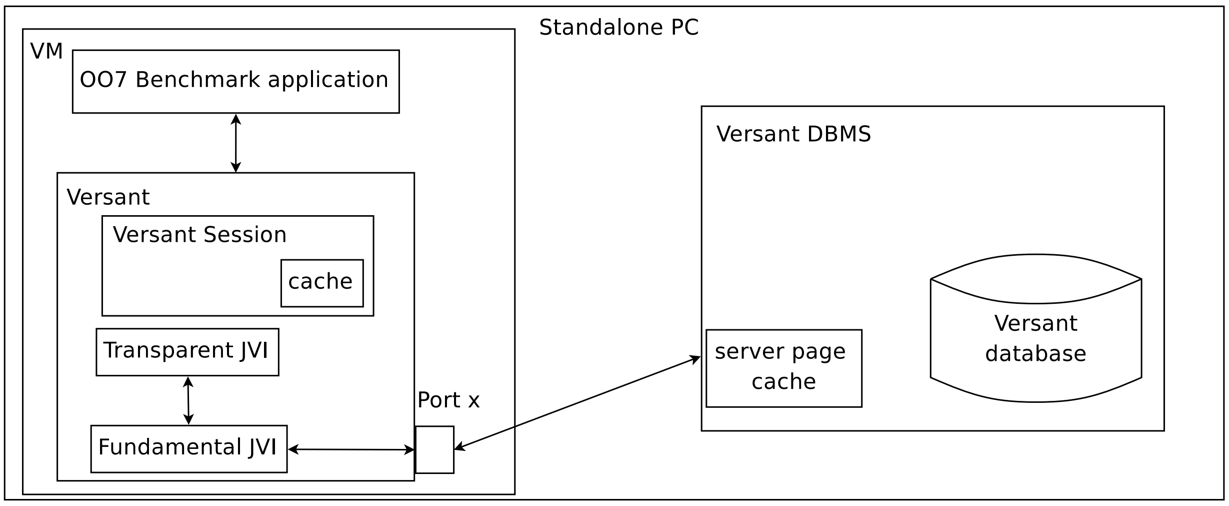


Figure 4.5: Architecture setup of Versant

4.4 Testing environment and versions used

The benchmark was run on a Dell Latitude D820 laptop with 2G of internal memory and a Intel Centrino 2.16 GHz *dual core* CPU (T2600). The 93G ATA Hitachi HTS72101 hard drive has a drive speed of 5,400rpm. The laptop has Mandriva Linux 2008.1 installed running kernel 2.6.24.5-laptop-2mnb. The system uses the *ext3* (third extended file system) [19] journal file system. Llanos [88] has shown, by using an TPC-C (Transaction Processing Performance Counsel) database benchmark implementation, that the type of Linux file system used, could have an impact on performance. Llanos [88] also shows that the use of single or multi-core CPU can influence the performance. For these reasons it is important to document what type of file system and type of CPU have been used.

The system was booted into run *init level 3* in Linux for the experiments. This was done to remove the X window [29] system and all the extra applications that use X.

The JVM, using Ant, was setup using the following parameters:

- `<jvmarg value="-server" />`. In Java 5.0 a new *feature* called "ergonomics" was introduced. It can be used to automatically tune and set performance parameters on the JVM by using already specified machine class profiles [1, 94]. The two classes are the *client* and the *server* class. The client class creates a client VM optimised for fast start-up time and a small footprint while the server class creates a VM optimised for fast execution speed [2]. For the study's benchmark runs, the server-class was selected.

- minimum and maximum heap settings:
  `<jvmarg value="-Xms700m" />`
  `<jvmarg value="-Xmx1100m" />`.

Furthermore, the laptop was not plugged into a network. Some Linux kernel processess were running in the background which could create a certain amount of overhead in the measured times. However, there does not appear to be any reason why such overhead would influence one persistence mechanism more or less than the other. All the benchmark operations were run on a standalone PC. *Future work* will run the benchmark operations on a dedicated machine while the databases are run on their own server machines.

Technology	Version
PostgreSQL	8.2.7
Hibernate	Hibernate 3.2.5_ga
PostgreSQL JDBC	8.2-505.jdbc3.jar
Hibernate JPA Annotations	3.3.0.GA
db4o	db4o-7.4.71.12224
Versant	7.0.1.4
Java	Sun Java 2 Runtime Environment Standard Edition (build 1.5.0_15-b04)
Ant	1.6.5

Table 4.2: Versions of the technologies used

4.5 Overall observations on the implementations

When creating an implementation of OO7 for a specific object database or ORM tool, there is bound to be uncertainty about the correctness of the model that is saved. There is, of course, no silver bullet – no automatic way to check the correctness - one has to rely on testing. Thus, when the Module (the parent class in the OO7 benchmark) and its entire hierarchy are saved, the ideal would be that there should be certainty that the whole tree is correctly saved and retrieved. Unit tests for all of the operations that modify the model are required to enhance confidence that these individual parts are working correctly. Although this is envisaged at a later stage, for the interim counters were inserted into the implementations to count the number of objects created. The number of persisted objects was then verified through direct database access.

Chapter 5

Results by Product

The chapter provides the OO7 benchmark results for *each* of the OO7 implementations created for Versant, db4o and Hibernate using them in their *"out of the box"* state. The results relating to the use of different *performance techniques* (indexing, caching and lazy loading) and their impact on the performance of the object databases and the ORM tool are shown and discussed in this chapter. All the implementations were run using the *small configuration* of the OO7 benchmark with *three connections.* The small configuration database consists of ±43 000 objects.

In Van Zyl et al. *[96]*[1], the in-memory implementation results, for traversals, were also reported. The in-memory implementation was a baseline to indicate the best possible times, when objects are accessed and stored in-memory without the use of a database. Objects were kept in-memory in Java collections and only OO7 traversal operations where supported. The operations for queries and modifications were never implemented for the in-memory version. The in-memory results showed that in-memory processing takes up an insignificant amount of time. This preliminary investigation did not reveal anything of significance in interpreting cross-product performance. Hence, it was considered that very little would be lost in leaving in-memory results out.

The next chapter, Chapter 6 contains detail comparisons between the object databases and the ORM tool.

For all the test configurations certain initial expectations existed. These expectations are:

- Lazy loading will be faster than eager loading.

[1]Note that an error was subsequently found in one of the calculations. An erratum describing the error and its impact was provided on-line [97].

- Lazy loading along with indexes should be faster than just using lazy loading.

- Using indexes should make queries faster.

- Queries should be faster when using an ORM tool along with a relational database.

- Object navigation will be faster using an object database.

For each of the implementations results are provided for the time to create the database, traversal times, query times, insert and delete times. It is important to note that to *create* the database involves creating new objects and *linking them together*.

Queries are used to find objects in the model that matched certain query criteria. When the query results are being investigated it helps to keep the following classification of the OO7 queries in mind:

- Query 1 refers to queries on one table/class type where object ids are used to match.

- Query 2 and Query 3 refer to queries that find objects in a certain range.

- Query 7 refers to a query that finds all the objects of one type of class. In this case find all `AtomicPart` objects.

- Query 4 and Query 5 refer to queries that find objects of type A and then traverse their one-to-many associations. Query 4 also involves creating random object ids than must be found.

- Query 8 refers to a query that find objects of type A and performs a "join" with another object B, using ids to do the matching.

All of the queries involve iterating through the results returned and sending back the number of objects found. Query 6 was not in the original OO7 results as it did not provide any insight into the performance. For this reason Query 6 was also not included in the current study. Future work could include it.

The sizes of the databases for PostgreSQL, db4o and Versant are reported in this chapter. They will be compared with each other in the next chapter to find out which has the *smallest* database.

All the graphs rely on a logarithmic scale for times. Logarithmic scale was used because some of the times are very high, near a 1000 or 10 000 seconds, while others were very low near 0.001 seconds. These low times could not be seen using a normal scale and for this reason a logarithmic scale was selected.

5.1 Hibernate on PostgreSQL results

This section discuss the results for the Hibernate implementation of OO7. The Hibernate test configurations are discussed in detail in section 4.2.4. The results for Hibernate are shown in Figures 5.1, 5.2, 5.4, 5.6 and 5.7.

5.1.1 Creation

Creating the OO7 database using *eager* loading took too long. To create the database using eager loading with indexes could take about 2 days and to create a database using eager loading without indexes could take up to 10 hours. This situation was investigated and it was noticed that there was a lot of garbage collection happening during database creation. The PostgreSQL server was also working very hard. In the end the database was created using the *lazy* loading Hibernate XML mappings files. When the benchmark *operations* were *run*, the correct eager loading Hibernate XML mappings files were used.

From Figure 5.1 it is clear that using Hibernate with annotations and named queries is slightly faster than using Hibernate XML mapping. Named queries were added because it was found that using just annotations was actually slower than using Hibernate XML mapping files. “Annotations are compiled into the byte-code and read at run-time (in Hibernate’s case on start-up) using reflection” [74]. When Hibernate is started and the session factory is created the annotated classes are read which seems to create a slight overhead. When named queries are added it improves the overall speed of Hibernate.

Using indexes does not slow down the creation times. This was unexpected.

The size for the OO7 small databases in PostgreSQL is ± 71 MB.

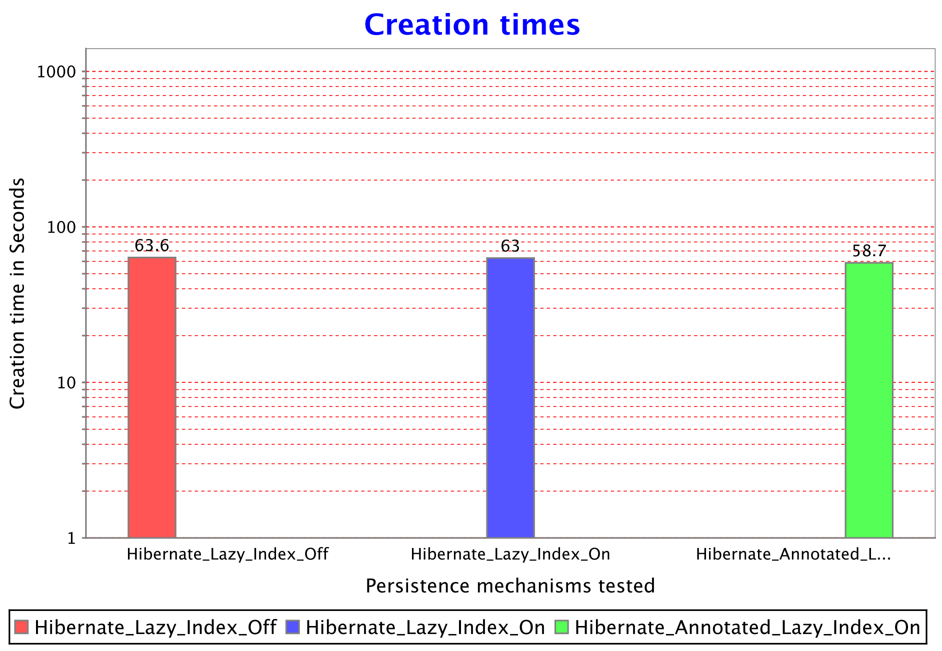


Figure 5.1: Hibernate database creation results

5.1.2 Traversals

Figure 5.2 clearly shows that using *lazy loading* is faster than *eager loading* in all the *cold* runs. In the *hot* runs however, the eager loading and lazy loading configurations are very close together. A possible reason for this is that the *cache* is now populated and being used in the hot runs. Thus, even though eager loading the object tree into the cache for the first time is slow, it apparently assists in the object traversal or navigation of the tree in the later iterations.

As stated in the previous chapter a cold run is a run where all the caches are empty and a hot run is where objects have been cached.

The *average speed improvement percentage,* due to the *cache* between the cold and hot runs for traversals, was calculated using the following formula:

$$\frac{(coldTime - hotTime)}{coldTime} \times 100$$

for each operation and then taking the average over all traversals.

Figure 5.3 shows that on average a 91% speed improvement is obtained due to the cache for all test configurations involved in traversals. The *first level or session cache* provides this improvement because it caches the actual objects in the first run and the same objects are used in later iterations, using the same session. There is thus no need to request the objects from the database again [39].

The *speed* improvement, due to the cache, provided for *eager* configurations when running *traversals* is very high and very low when running *queries*. Figure 5.3 also shows that the cache in general improves the speed of the runs for traversals more than for queries. The possible reason for this is that queries involve more random objects while traversals involve traversing the same tree of objects.

A possible rule of thumb when traversing and when it is known that the same group of objects will be traversed, is to use eager loading. This of course depends on how large the object tree is and on the available memory.

Hot traversals *Trav2b, Trav2c, Trav3b,* and *Trav3c* are very interesting. These traversals involve a large number of objects: 43700 for *Trav*2b and *Trav*3b, and 174800 for *Trav*2c and *Trav*3c. Figure 5.2 shows that *eager loading* with no indexes performs slightly better than *lazy loading* in the *hot runs* for these traversals. This was not expected because it was assumed that using lazy loading for large number of objects would always perform better. A possible reason for why the eager loading run is faster than the lazy, is that the database is so small, that most of the database could easily be cached in memory on the first run with eager loading. When using lazy loading there is still a check to see if the object is null or already loaded into memory.

When a larger database which cannot fit into memory is used, it is expected that lazy loading should be faster than eager loading. If lazy loading is used for such a small database it creates a slight *overhead* in the *hot* runs. For these traversals (*Trav2b,Trav2c,Trav3b,* and *Trav3c*) the graph also indicates that when using an index, the traversal runs are slower.

Indexes are used to improve query times to find a specific object. When traversing, the first step is to find a module with a specific ID and then traverse the rest of the tree from that root. An index will help on the first query for the module, but not for the rest of the object traversals.

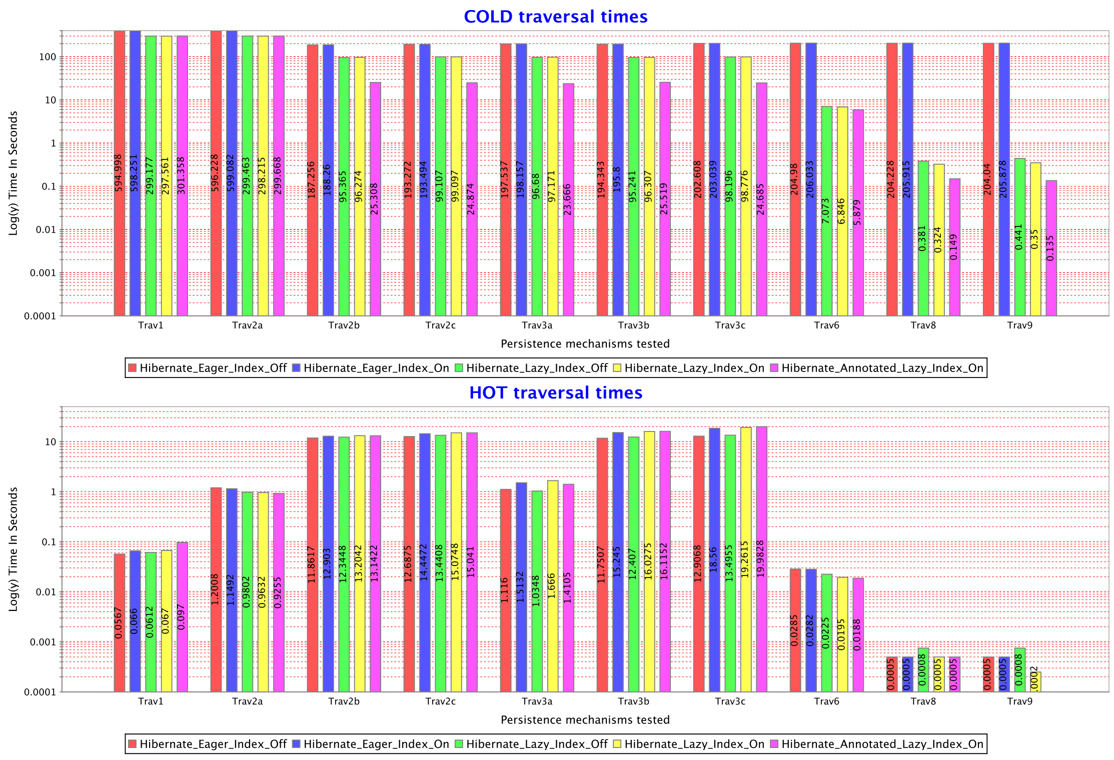


Figure 5.2: Hibernate cold and hot traversal results

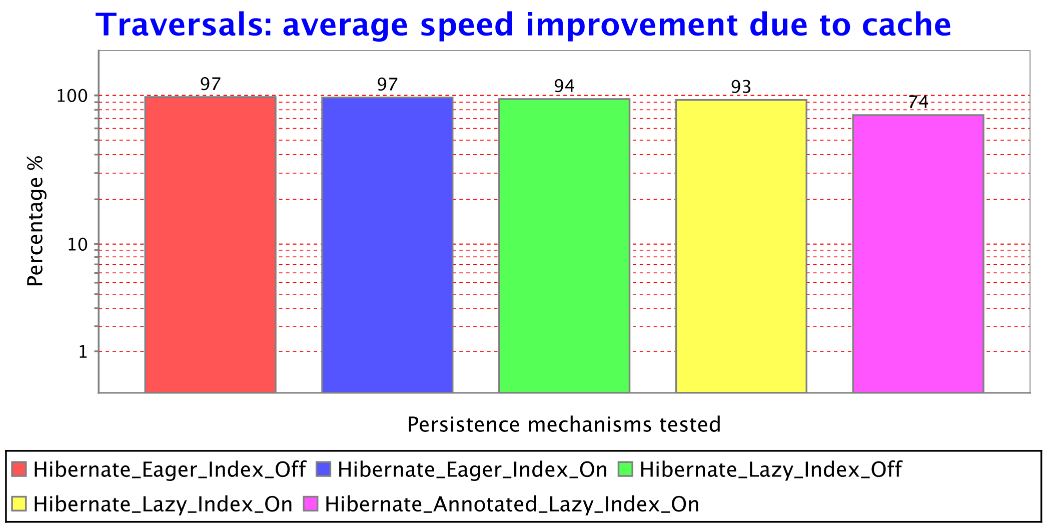


Figure 5.3: Hibernate average speed improvement due to the cache for traversal runs

5.1.3 Queries

The OO7 benchmark was created to test persistence performance with respect to objects. In order to achieve this goal, it emphasises object navigation.

Most of the queries in OO7 are written in an object navigational way, meaning that specific objects are retrieved or *queried* for first, then their associated objects are navigated. This means that when a query is executed, a result set is returned, and then iterated through, accessing or navigating either the objects properties or its associated objects.

While most queries only iterate through properties, Query 8 also accesses associated objects. Figure 5.4 shows that the run-time for Query 8 is 3056 seconds when using lazy loading with indexes and 8980 seconds when using eager loading and no indexes, which is very slow. Query 8 suffers from the *n+1 select problem* [39]. To understand the problem better, one needs to understand what Query 8 is doing. Query 8 states:

> *Find all pairs of documents and atomic parts where the document ID in the atomic part matches the ID of the document. Also, return a count of the number of such pairs encountered.*

In OO7 this means that first, all 10 000 `AtomicPart` objects are found, which are then iterated through finding the associated `Document` object. When iterating through the 10 000 `AtomicPart` objects, *another query per* `AtomicPart` is executed on the database to find the associated `Document` object. This is known as the n+1 select problem.

Bauer and King [39] discuss a similar n+1 problem related to lazy loading in detail and provide solutions for this problem, and also mention that it is important to *minimise* the amount of queries sent over the network.

From on-line [3] and personal discussions about why Query 8 was slow in Van Zyl et al. *[96],* a modification to Query 8 was suggested[2]. The modification was to use a *join* query to limit the number of queries sent to the server. Other suggestions included using indexes and lazy loading. The new *join* query in HQL is:

from Document doc, AtomicPart part where doc.docId = part.docId

While the new *join* query was a bit faster there was still another problem. This had to do with *eager* loading. If eager loading is on, and the 10 000 `AtomicParts` are returned, the 30 000 associated `Connection` objects are also

[2]These suggestions have been made by William Cook from the *Department of Computer Sciences, UT Austin http://www.cs.utexas.edu/~wcook/*

retrieved. This meant that for each `AtomicPart` *another join* was executed with the table containing the `Connections`. This is unnecessary as the connections are not used in Query 8. By enabling *lazy loading* Query 8 increases in speed and with the new *join query* the time goes down to about 3 seconds in the cold runs and 1 second in the hot runs. The new Query 8 is called Query8Join and both the original and the new query are shown in Figure 5.4.

It is clear from Figure 5.4 that lazy and eager loading have an impact on the performance of *all the queries.* Query runs, using eager loading, are slower than runs where lazy loading is used.

When a *large* number of objects are queried (Query 7 and 8) and Hibernate is used with annotations and named queries it is slightly slower than the Hibernate XML configurations.

The use of indexes does improve the querying speed. Query 1 involves finding `AtomicPart` objects by matching on an “ID” on which there is an index. Looking at the hot query graph, it is clear that the index does help to improve the query speed.

Query 7 is also querying 10 000 `AtomicParts` but is slightly faster than Query 8 because there is no join or n+1 query involved.

Query 5, which queries for 199 objects, is very slow because it also suffers from the n+1 select problem and could also be rewritten using a join query.

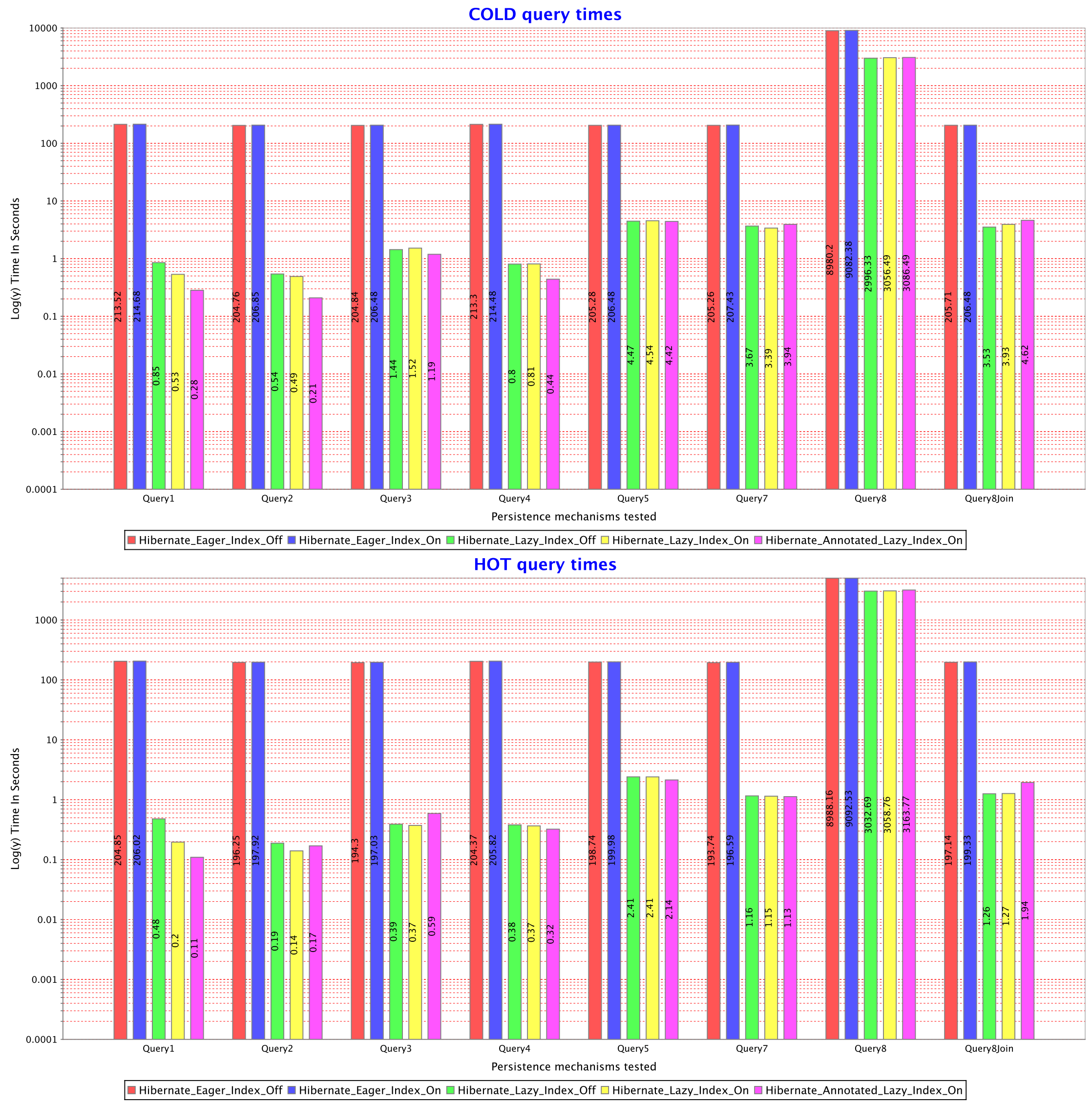


Figure 5.4: Hibernate cold and hot query results

Figure 5.5 shows the speed improvements provided by the cache between the cold and hot runs for queries. When lazy loading is turned on the average speed improvement provided by the cache is 50% and for eager loading the average is 4%.

It is important to note that Hibernate provides a *query cache* that can be used to cache query result sets [39, 76]. The query cache was not used in these configurations. Bauer and King [39] also state that the query cache provides minimal improvement in most use cases. Because the OO7 queries are repeated 5 times, using the query cache should improve the speed for the

OO7 queries in the query test runs.

Figure 5.3 and Figure 5.5 show that cache helps *traversal* operations more than *queries*. For traversals the average speed improvement is 91% and for queries it is 31%.

This is probably because once the object graph being *traversed* is in memory, it is very fast to traverse the references and there is also no need to ask for the object from the database. The OO7 *queries* on the other hand involve mostly *querying for random generated object IDs* and sending each query to the database for each random selected object ID to find the matched object. This implies that even if the object is cached in the first iteration, the second iteration and query will probably need to find a different random selected object. There is a high probability that the second object will not be in the cache.

From the results it is clear that the normal Hibernate session cache does not improve the query speeds as much as for traversals. If the *query cache* was enabled the cache use would have improved. Queries using eager loading will be slower because objects are brought into the cache that will not be used and this extra retrieval makes it slower.

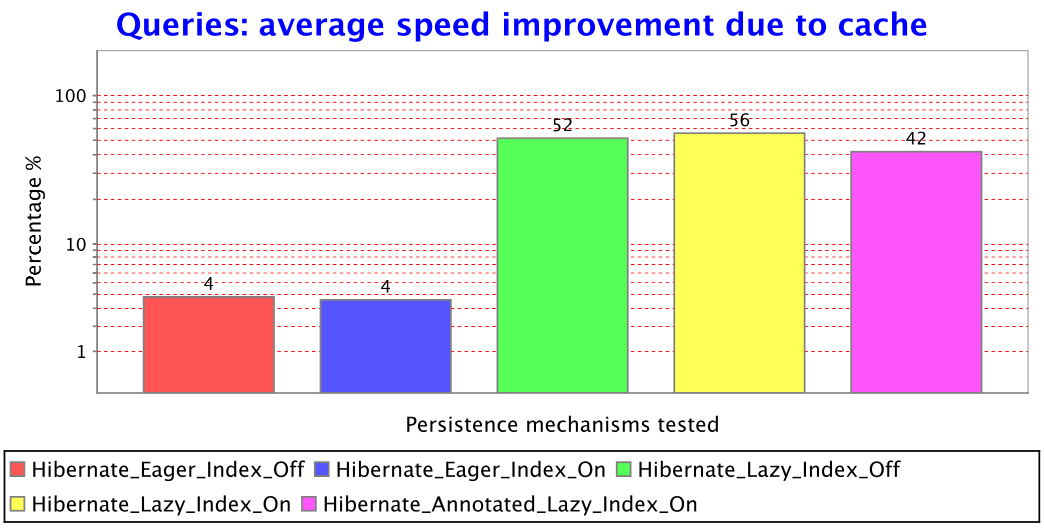


Figure 5.5: Hibernate average speed improvement due to cache for query runs

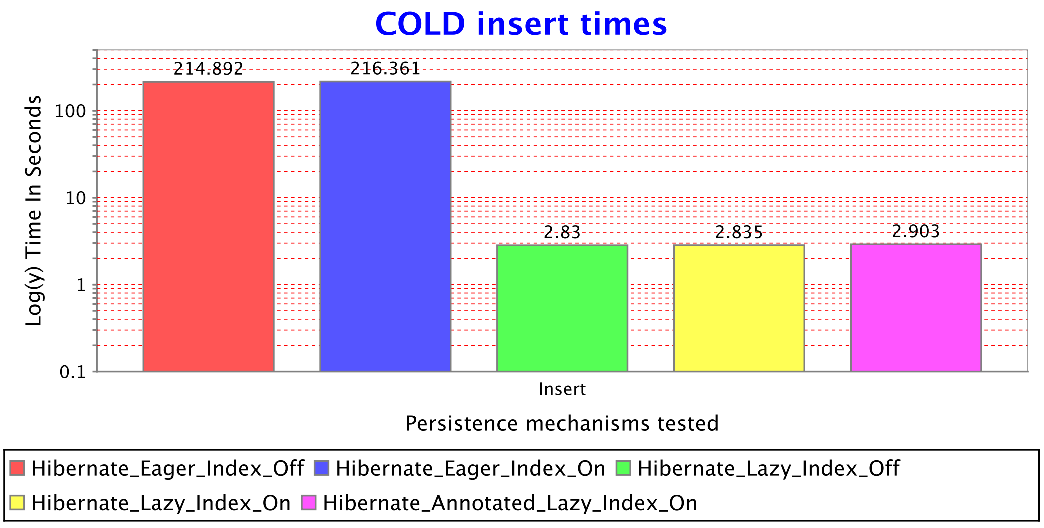
5.1.4 Modifications: Insert and delete

Figure 5.6 and 5.7 clearly show that using *lazy* loading during inserts and deletes is faster than *eager* loading. During insertions, objects are added to these collections and during deletions objects are removed from these collections. This means that the objects and their associated object trees are unnecessarily loaded into memory using eager loading which makes it slower.

When inserting objects in the cold runs it is observed that the use of indexes is slower. This is because there is a slight overhead when the index is created for the newly inserted object. Indexes also make deletes a bit slower as the index must be removed as well.

During the hot insert runs having an index does not slow down inserts as the cache kicks in and helps to improve the insert times by caching the object collections to which the new objects are added.

The Hibernate annotation's configurations times are very close to the Hibernate XML mapping file configurations times.



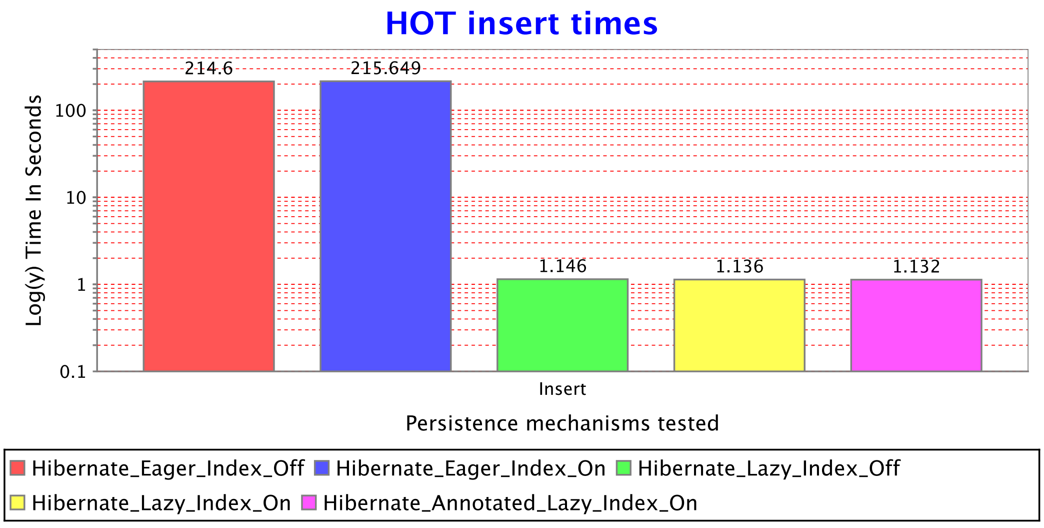


Figure 5.6: Hibernate cold and warm insert results

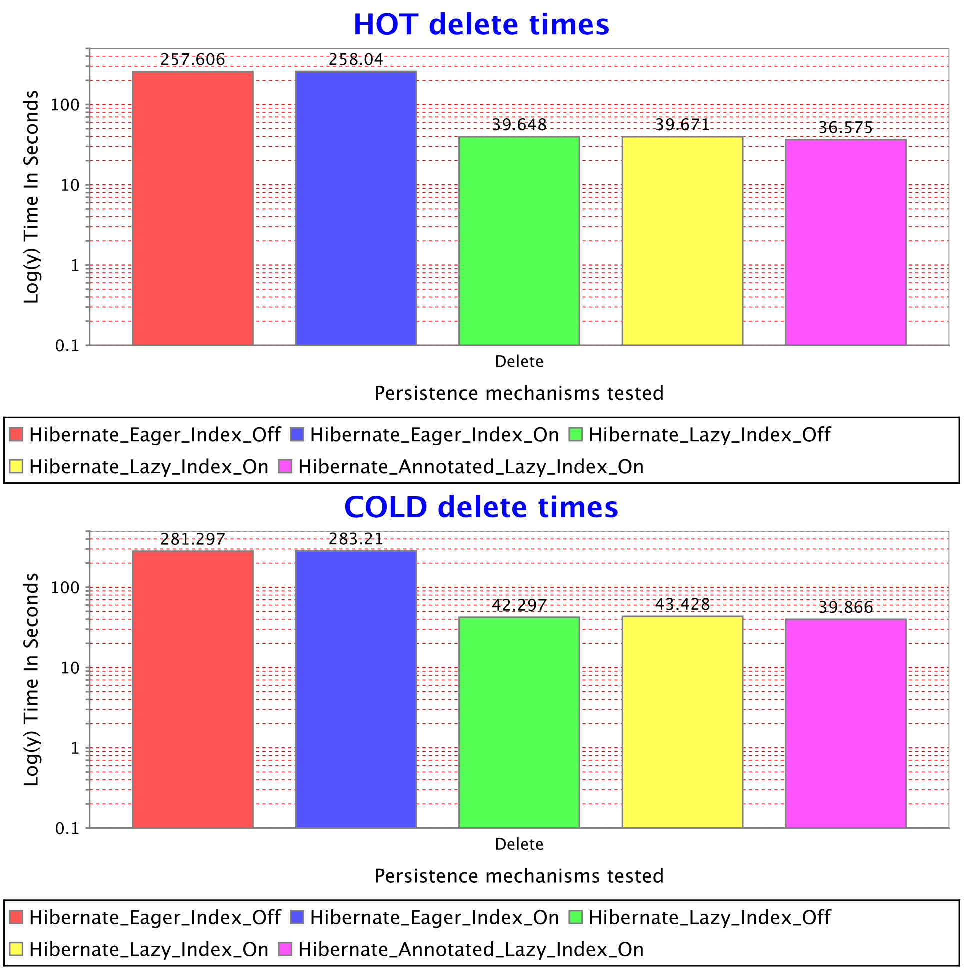


Figure 5.7: Hibernate cold and hot delete results

Table 5.1 shows the average speed improvement provided between the cold and the hot runs by the cache. The cache improves the insert runs on average ± 36%. For delete runs the average is very low at 8%. When lazy loading is used the cache provides on average a 60% speed improvement for inserts.

In Hibernate inserts are ± 39 seconds faster than deletes.

Configuration	% Improvement for Inserts	% Improvement for Deletes
Eager loading and indexes off	0.14	8
Eager loading and indexes on	0.33	9
Lazy loading and indexes off	59	6
Lazy loading and indexes on	60	9
Using annotations with Lazy loading and indexes on	61	8

Table 5.1: Hibernate average speed improvement due to the cache for modification runs

5.2 db4o results

This section discusses the results for the db4o implementation of OO7. db4o was run in networked mode (a network server and client-server mode) with the client and server each in its own VM. A standalone configuration was included to investigate the performance difference between using *networked client-server mode* versus *standalone mode* (*embedded* database). One of db4o's strong points is that it runs as an embedded standalone database. The results obtained will be discussed in the following sections.

5.2.1 Creation

The creation times in Figure 5.8 show clearly that eager runs are about 4 times slower than the lazy runs when running in *networked client-server mode*.

During a subsequent review of the results by db4o they recommended turning `cascadeOnDelete` off when performing inserts and updates during the database creation and linking process. It was found that there existed a known bug that influenced the inserts [64] and created an extra overhead. Each time an object was inserted an unnecessary callback was called. This bug exists in the version of db4o that is currently being used (db4o-7.4.71.12224) for this study. In later versions this bug has been fixed. It was decided to add the boolean variable `SettingsUtil.useCascadeOnDelete`. This was used to turn cascade on deletes *on* only when performing *deletes*. This solved the overhead. It was not possible to upgrade to a newer production version of db4o because *then* it would have only been fair to upgrade Hibernate and Versant as well. Future work will use the newer versions of these products.

The time to create the database using lazy loading in *standalone mode (embedded mode)* was 13433 seconds before fixing the bug. After the bug was fixed the time to create the database was around 7780 seconds. This is very slow. This could be related to an extra *cascade on update depth* setting that was needed on the `CompositePart` class to store the whole model when running in standalone mode. Because db4o is an object database with a focus on embedded devices and embedded applications on these devices [62] it was expected that the standalone mode (embedded database) of db4o would be faster than than the *networked client-server mode* for such a small database.

The db4o OO7 small database sizes are about 6 to 9 MB.

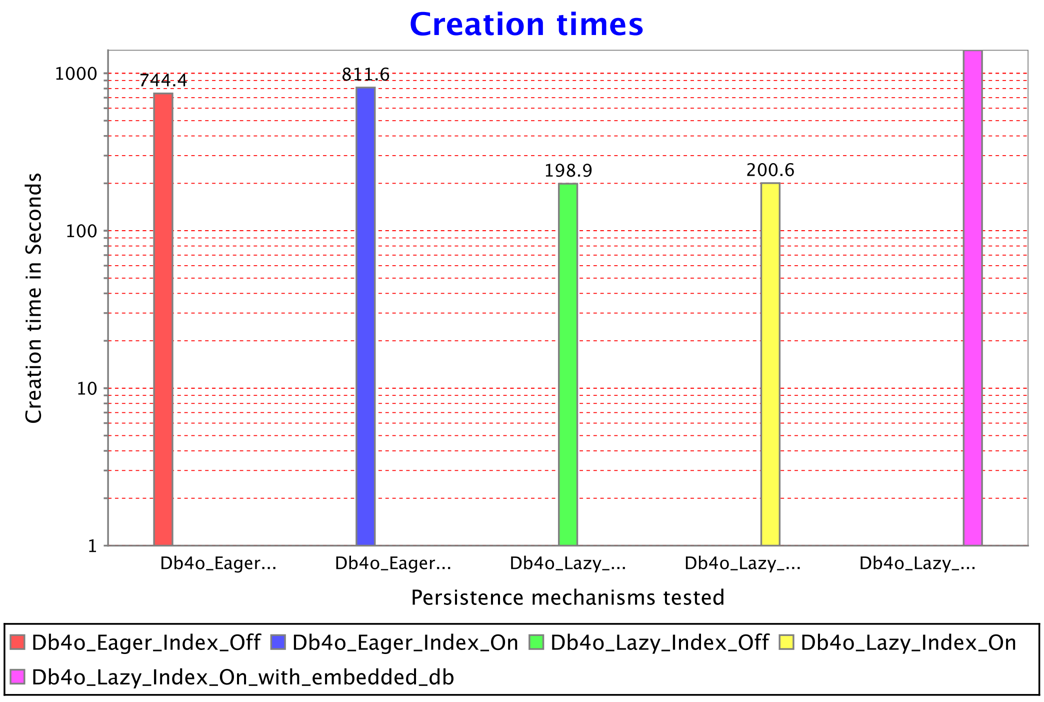


Figure 5.8: Db4o database creation results

5.2.2 Traversals

Traversal runs traverse the object tree in an object oriented fashion. The results are shown in Figure 5.9.

In the *cold runs* it seems that when a *small amount of objects* are traversed, like in traversal *Trav6* (2185), that the *standalone* and *networked client-server lazy* configurations are faster than the *eager* client-server ones. When the traversals involve a larger amount of objects, like traversals *Trav2b (43740), Trav2c (174 960), Trav3b (43740)* , and *Trav3c (174 960)* the *eager* loading runs are faster than the lazy loading configurations. Eager loading loads the whole tree into the *weak reference cache (*discussed in section 4.1.2*)* on the *first* iteration so it was expected to be slower in the cold run, but the fact that it seems to be faster than the lazy runs for larger data sets like *Trav2c* and *Trav3c* in the cold run comes as a surprise. In the lazy runs there could be an overhead on the first cold run with all the calls to activate some of the larger collections, which causes an overhead as it needs to request these objects from the db4o server and then put them into the cache. Eager loading uses one call to load the whole tree and lazy loading needs many calls.

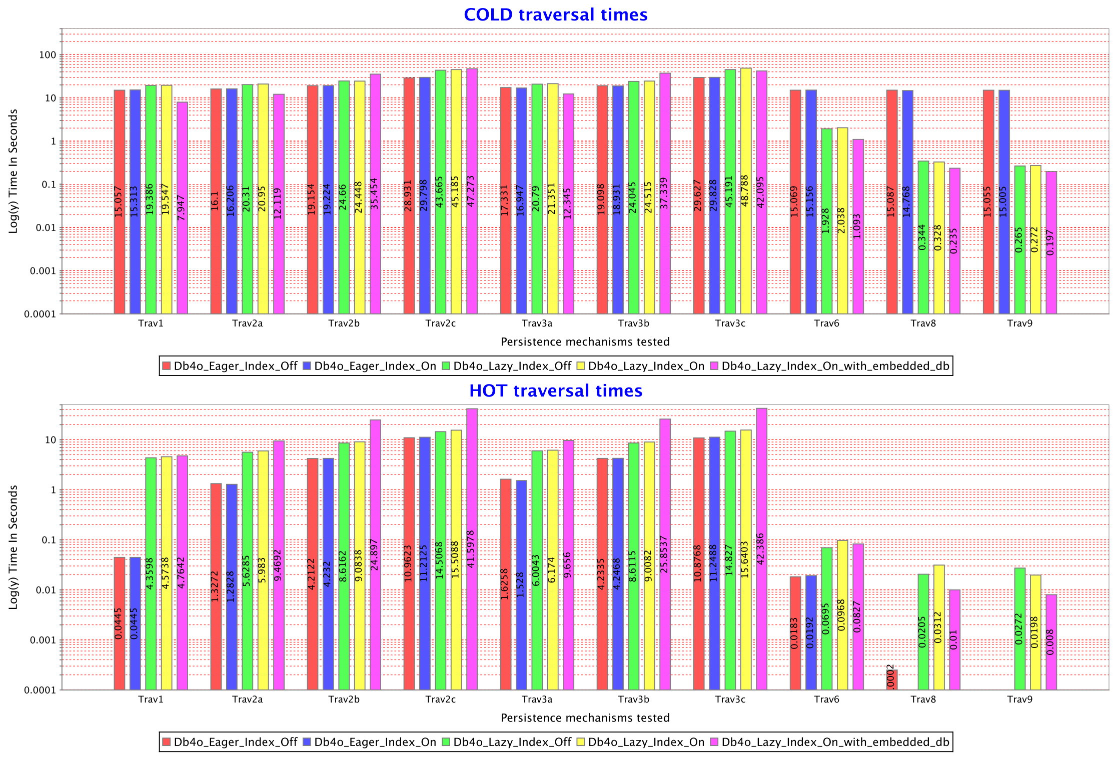


Figure 5.9: db4o cold and hot traversal results

In the *hot runs* the eager loading configurations are *all* faster.

Trav2a, Trav3a and *Trav6* traverse 2185 *"root atomic parts".* The main difference is that *Trav6* does not involve updates to the objects while being traversed. The fact that objects are being changed and saved in *Trav2a* and *Trav3a* explains why they are much slower than *Trav6.* For example the db4o lazy loading configurations take about 20 seconds when performing *Trav2a* and *Trav3a* in the cold runs, but only 2 seconds *for Trav6.*

Traversals *Trav8* and *Trav9* involve traversing a very small amount of objects and are very fast.

Hibernate differs from db4o in that the eager runs was slower in the *cold runs* and then slightly faster or very close to the lazy loading runs in the *hot runs*. In db4o the eager runs are in most cases faster than the lazy runs in the cold runs. Hibernate will be compared with db4o in the next chapter.

The speed improvements due to the cache when traversing is shown in Figure 5.10. It seems that standalone mode, using lazy loading, performs better in cold runs than in the hot runs and that its speed improvement due to cache is less. For the standalone mode the cache only provides a 44% improvement while for the other configurations the improvements are between 86% and 75%. The eager runs have a slightly better speed improvement when a cache is used in traversals.

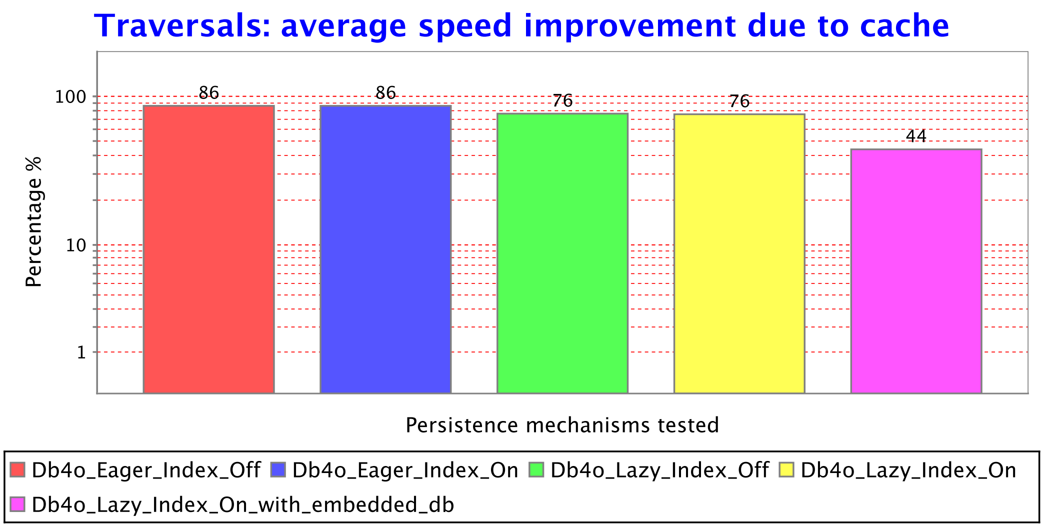


Figure 5.10: db4o average speed improvement due to cache for traversal runs

The db4o networked client-server mode is faster than the standalone mode in the hot runs. It is possible that db4o, when running in client-server mode, performs caching at the server as well. That could improve the client-server performance.

As with Hibernate, *indexes* are used to improve queries to find a specific object. When traversing, the first step is to find a module with a specific ID and then traverse the rest of the tree from that root. Thus an index will help on the first query for the module, but not for the rest of the object traversals.

From the foregoing it seems that eager and lazy loading as well as the db4o mode have a large impact on the performance. As seen in Figure 5.9, *Trav1,* eager loading can make a difference of as much as ± 98% in the hot runs. Figure 5.9, *Trav2c,* shows that db4o in networked client-server mode can be between 63% and 73% faster than db4o in standalone mode in the hot runs.

5.2.3 Queries

Query results for db4o differ quite dramatically from traversal results. The results are shown in Figure 5.11.

Queries run in db4o standalone mode are faster in cold runs than queries run in client-server mode. It is also faster in most of the hot runs. This could be related to an *overhead in communication* between the client and the server and the fact that some of the queries, like Query 8, also suffer from the *n+1 select* problem—meaning that there are extra query calls to the server. Using a join like query, or batching query calls, or creating look-up tables (such as views in SQL) could improve these query times.

While *eager* runs are faster for traversals they are actually *slower* for *queries* in hot and cold runs. All the lazy query runs are faster than the eager query runs. The issue of eager runs being slower with regards to queries is similar to the issue discussed in Section 5.1.3 for Hibernate queries. As stated, in for example Query 8, all the associated object collections are loaded when eager loading is used. This extra loading makes queries slower when using eager loading. Thus, when a tree is eager loaded and this tree is actually traversed, eager loading and the cache helps to improve the speed. However, if the tree is loaded and not used, it creates a huge overhead as can be seen when running queries.

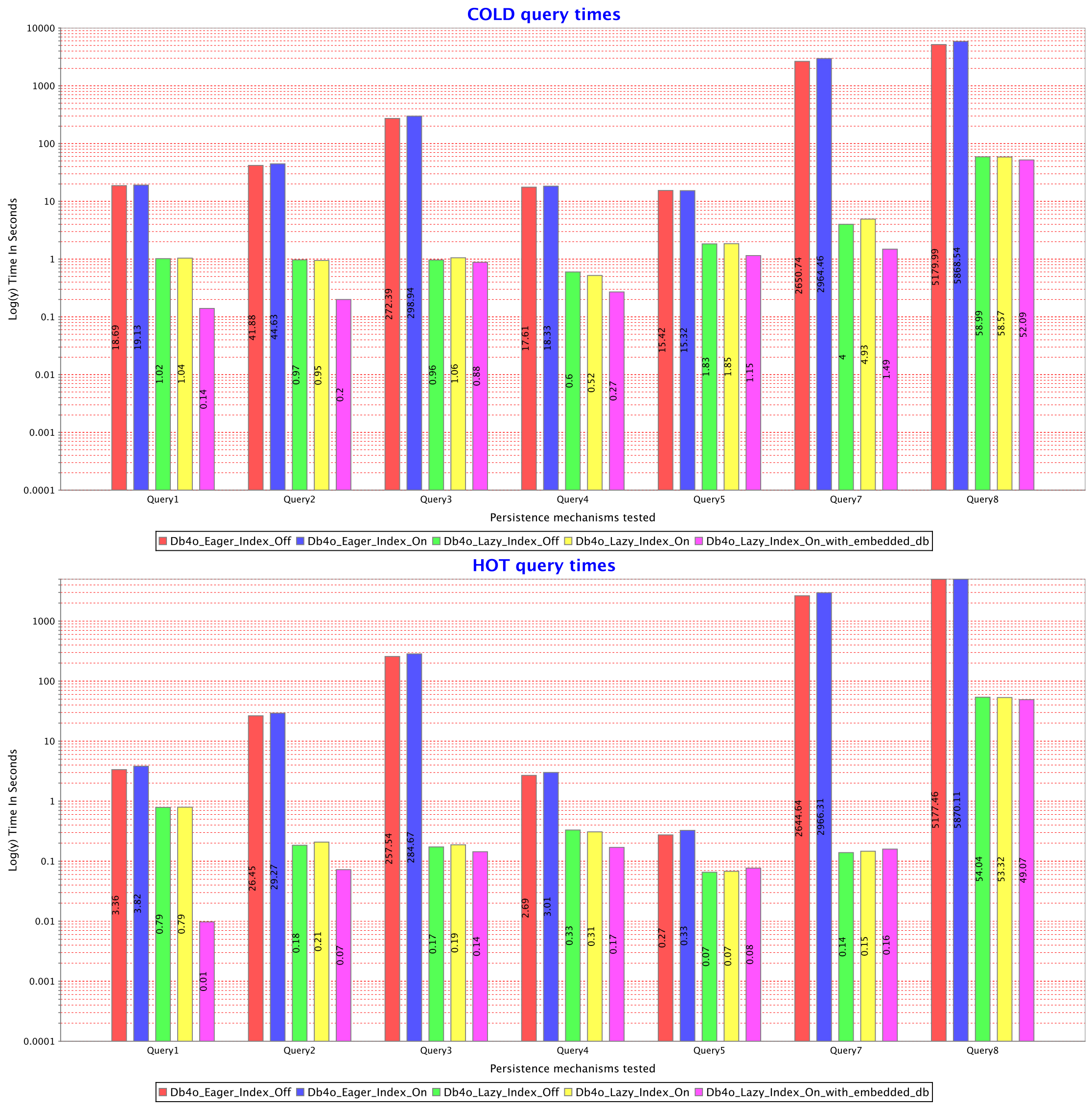


Figure 5.11: db4o cold and hot query results

Query 1 involves finding 10 `AtomicPart` objects by matching on an "ID" field on which there is an index. Unlike in Hibernate, the index in db4o does not improve the query speed. There is a slight overhead when using the index. For example the db4o eager loading configuration performs Query 1 in 3.36 seconds and in 3.82 seconds when the index is turned on.

Again, Query 7 is also querying 10 000 `AtomicParts` but is slightly faster than Query 8 because there is no "join" or n+1 query involved. Query 7 is performed in 4 seconds and Query 8 in 59 seconds when the db4o lazy loading configuration is used.

Query 5 which is querying 199 objects is actually faster than Query 2 (100), Query 3 (970) and Query 4 (42). Query 5 which is an n+1 type query is very fast in db4o, probably because of the caching and the fact that it involves object traversals as well.

Looking at the speed improvements due to cache in Figure 5.12 it is clear that in standalone mode the cache provides a ± 67% improvement. db4o in networked client-server mode on average only provides a ± 53%.

Figure 5.12 shows that the *cache* does not improve the eager runs (on average ± 43%) as much as the lazy runs (on average ± 65%).

Figure 5.10 and 5.12 shows that the cache helps to improve traversals speeds (on average ± 81%) more than queries (on average ± 53%) for the *networked client-server mode*. In eager query runs, times for large queries like Query 8 and Query 7 do not dramatically improve when the cache kicks in during the hot runs.

A possible and expected conclusion is that the *cache* improves traversal operation times and that objects used in queries are not cached.

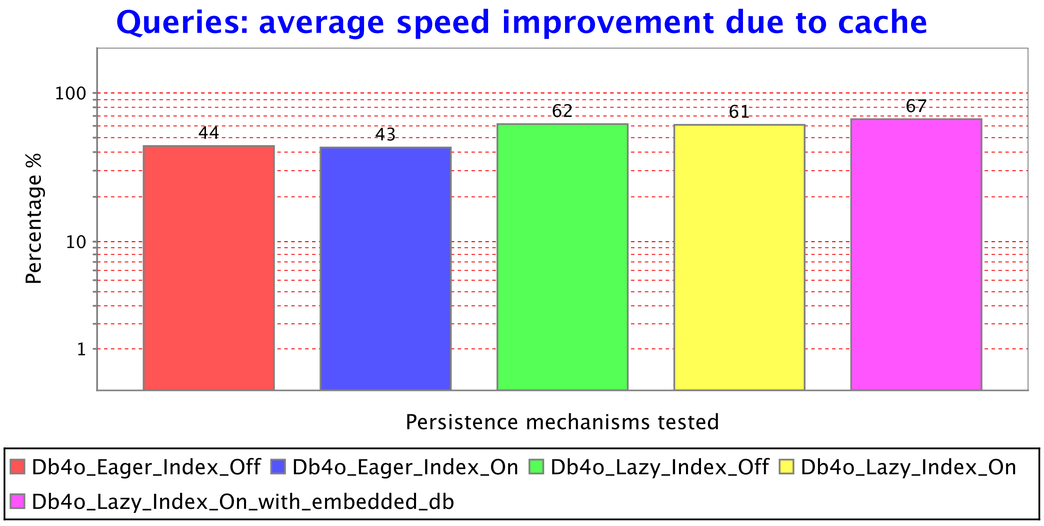
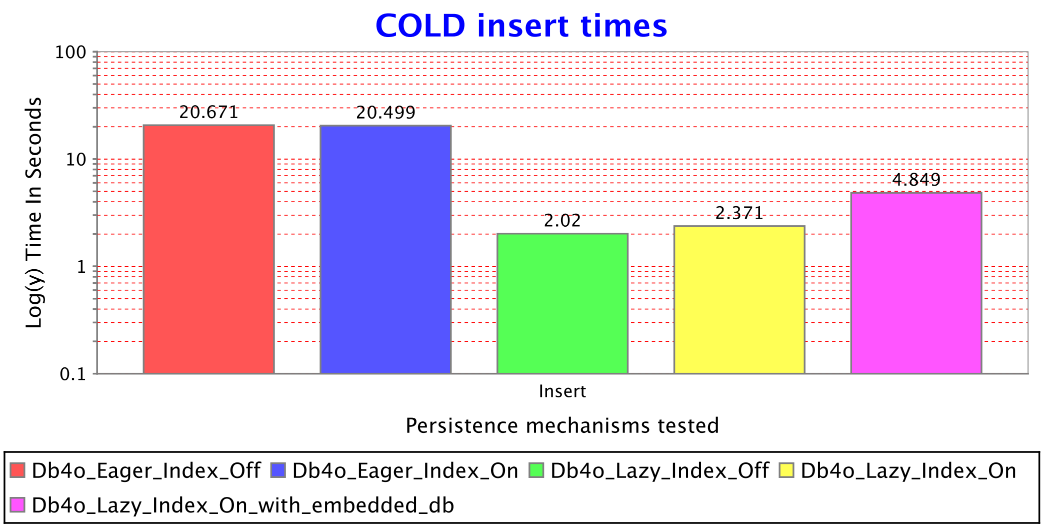


Figure 5.12: db4o average speed improvement due to cache for query runs

5.2.4 Modifications: Insert and delete

The db4o results for modifications are shown in Figure 5.13 and 5.14. The modification results are similar to the Hibernate results in that using lazy loading during inserts and deletes is faster than eager loading. *Eager* loading is *slower* because when inserting and deleting, the database is queried for objects and then *all* their associated collections and objects are brought into memory which creates an overhead. Objects are then removed or added to only some of these collections.



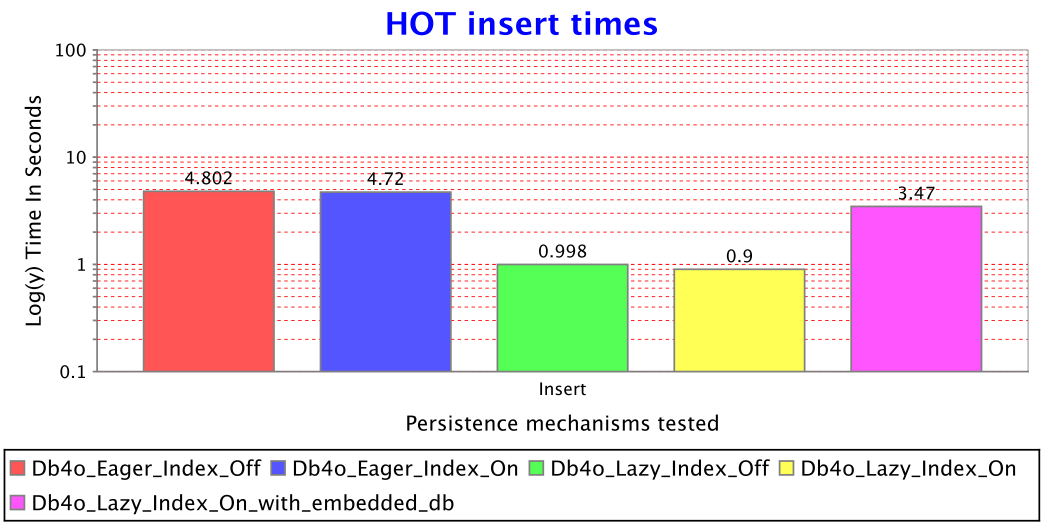


Figure 5.13: db4o cold and hot insert results

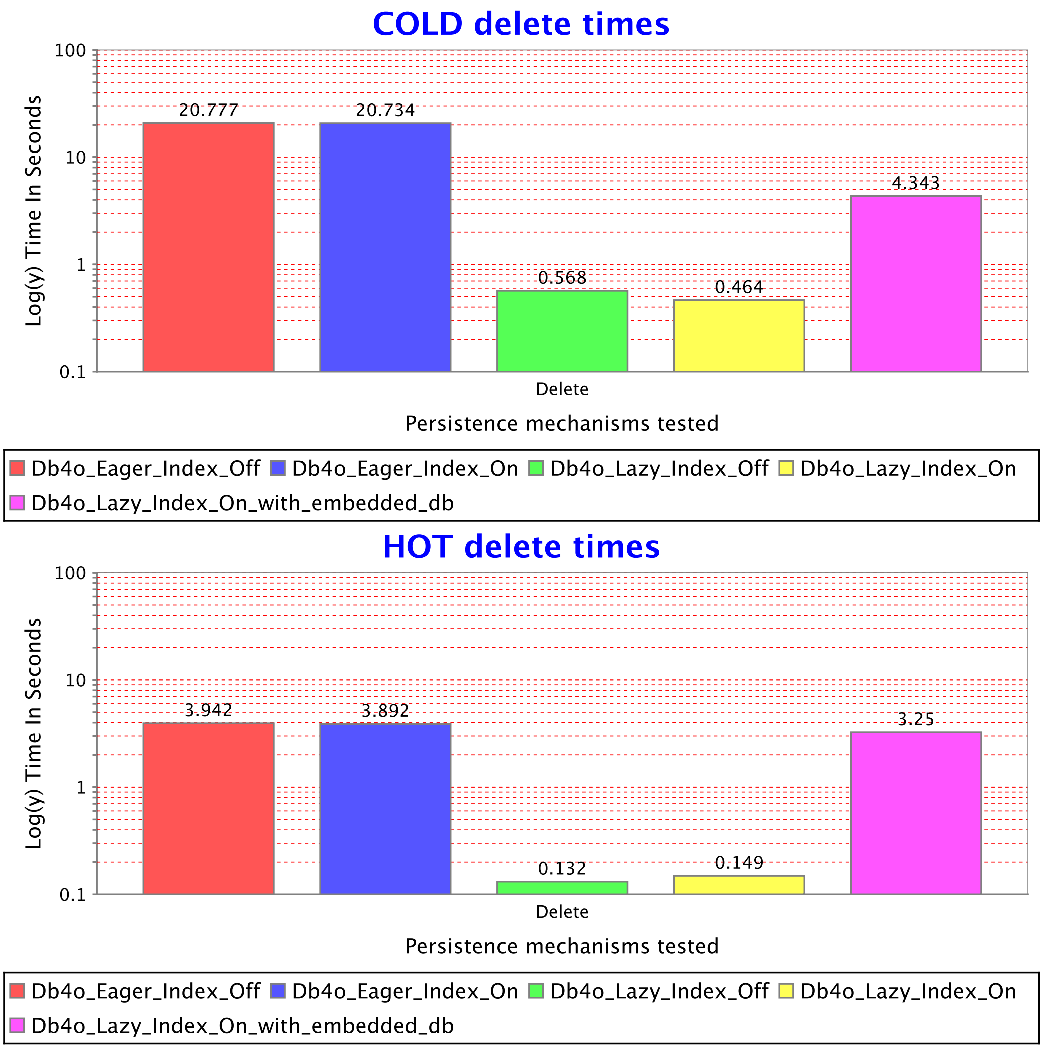


Figure 5.14: db4o cold and hot delete results

Having indexes turned on or off does not seem to influence the modification times in db4o. It was expected that if indexes are used that there should be a slight overhead when an object with and index field are added or removed from the database [69, 92, 106,120]. Kemper and Moerkotte [82, 70] state that while indexes improves query speed it slows down inserts, updates and deletes. This is because the index structure being used needs to be updated. They also state that indexes should be used when the application involves more read/query access. If the application is more dynamic and involves many updates, inserts and deletes then less indexes should be used as they create a overhead.

During inserts and deletes there are only ± 600 objects being added and

Configuration	% Improvement for Inserts	% Improvement for Deletes
Eager loading and indexes off	77	81
Eager loading and indexes on	77	81
Lazy loading and indexes off	51	77
Lazy loading and indexes on	62	68
Lazy loading and indexes on in standalone mode	28	25

Table 5.2: db4o average speed improvement due to cache for modification runs

removed. It seems that this low number of objects does not cause the db4o index structure any problems. It would be interesting to find out if there is any overhead when running the larger OO7 benchmark databases. Future work will investigate this.

Overall inserts and deletes speeds are very close together.

Table 5.2 shows the average speed improvement provided by the cache between the cold and the hot runs. In the presence of cache, eager runs improve on average by more than 77% while lazy runs improve on average in the range of 51% to 77%. Nevertheless, as can be seen in the graphs for hot insert and delete times in Figure 5.13 and 5.14, this improvement reduces the eager run times to just over 3 seconds, compared to the lazy run times which are in the order of 0.5 to 1 seconds.

The cache does not assist as much for the db4o embedded configuration.

5.3 Versant results

As stated in Chapter 4 there is no eager loading configuration available for the Versant implementation. After the first iteration of running the operations it was found that the times did not improve between cold and hot runs. After further investigation it was found that using `commit()` cleared the cache after each transaction was committed. This did not seem fair because db4o and Hibernate maintain their caches after committing a transaction. For this reason extra configurations were added using a `checkpointCommit()` which keeps the current cache associated with the session after committing a transaction. The results of using both are shown and discussed in the following sections. Keep in mind that for the Versant implementation some of the *persistent enhanced* collections were used.

5.3.1 Creation

The Versant creation times are shown in Figure 5.15.

As expected when adding indexes in Versant the creation times are slightly slower. Adding indexes in Versant is only done at the end when all the objects have already been created and saved to the database. Versant creation times are very fast when compared to db4o and Hibernate.

The Versant database size is ± 10MB for the database file and in total, with system files (log files and profile files), 15 MB

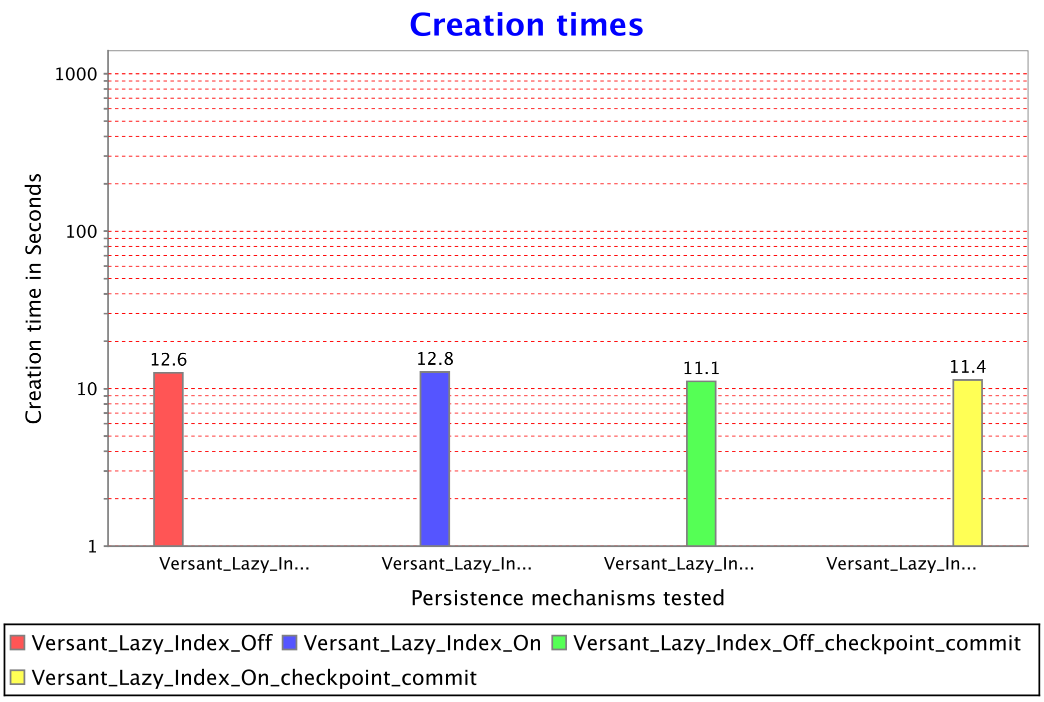


Figure 5.15: Versant database creation results

5.3.2 Traversals

Versant's traversal times are shown in Figure 5.16.

In the cold runs the times for the configurations are all very close, but in the *hot runs*, use of the *checkpoint commit* mechanism consistently improves the time for each traversal type.

Trav6, *Trav2a* and *Trav3a* traverse 2185 *"root atomic parts"*. The main difference between them is that *Trav6* does not involve updates to the objects which are being traversed. The fact that objects are being changed and saved explains why *Trav2a* and *Trav3a* are slower than *Trav6*. This is the same phenomenon that occurred in db4o.

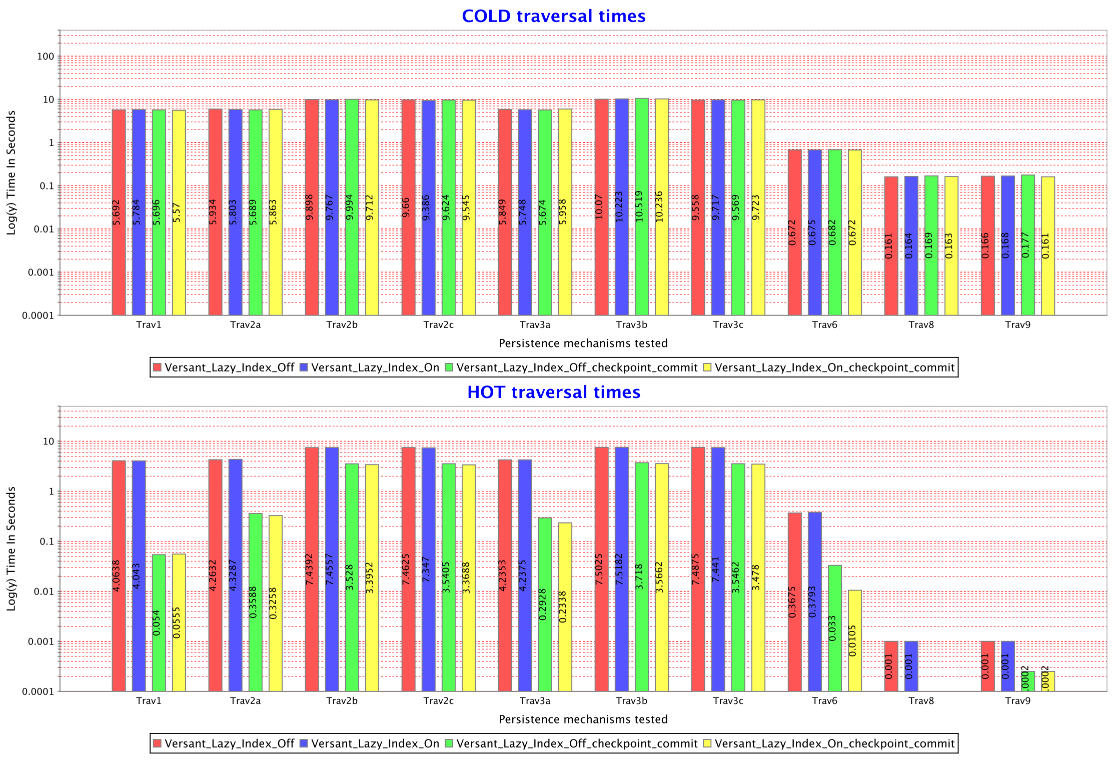


Figure 5.16: Versant cold and hot traversal results

Figure 5.17 shows the cache improvement during the hot runs. As expected checkpoint commits, which do not clear the cache, show on average, a 85% improvement. Using a `commit()` shows only a 41% average improvement as it clears the cache and then has to refill it on each run.

As stated in section 3.2.2 *many transactions* were used when running the benchmark operations.

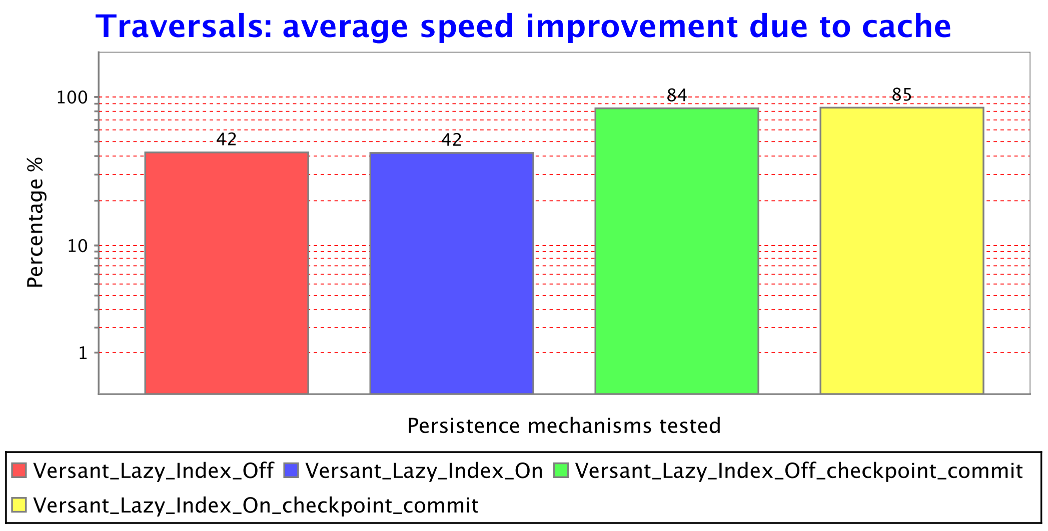


Figure 5.17: Versant average speed improvement due to cache for traversal runs

5.3.3 Queries

Figure 5.18 shows that *all cold run times* are about the same for all configurations tested. In the *hot runs* the configurations using checkpoint commits and the cache are faster for most queries. Query 8 is the slowest query and involves a join between `AtomicParts` and `Document` objects. Again this could be related to an *overhead in communication* between the client and the server and the fact that Query 8 suffer from the *n+1 select* problem—meaning that there are extra query calls to the server.

Query 1 involves finding 10 `AtomicPart` objects by matching on an "ID" field on which there is an index. With Versant, the use of an index does help to improve the query speed. It was mentioned before that Versant provides either *b-tree* or *hash table* index structures. For `AtomicPart`'s `buildDate` field a b-tree is used and for its ID field a hash table is used. This selection of different index structures was used because it was used in the original OO7 C++ implementation. Another reason that `buildDate` uses a b-tree index is that the field is used in value range queries. B-tree indexes are better suited for range type queries [100]. The *ID* field uses a hash table because hash tables are better suited to comparisons involving equality [100].

Query 7 is also querying 10 000 `AtomicParts` but is slightly faster than Query 8 because there is no "join" or n+1 query involved.

The time for Query 4, which finds 42 objects, is actually slower than the time for Query 2, which finds 100 objects. The difference is in the type of queries. Query 2 is a range query and Query 4 finds objects of type A and then traverse their one-to-many associations. Query 4 also involves creating random object ids that must be found. Because Query 4 finds these random objects all over the database it is slower.

Query 5 involves an n+1 type query. It iterates through `BaseAssembly` objects bringing in their associated private `CompositeParts`. Using a `checkpointCommit()` and the cache improves the speed here as this is basically a query and then a traversal.

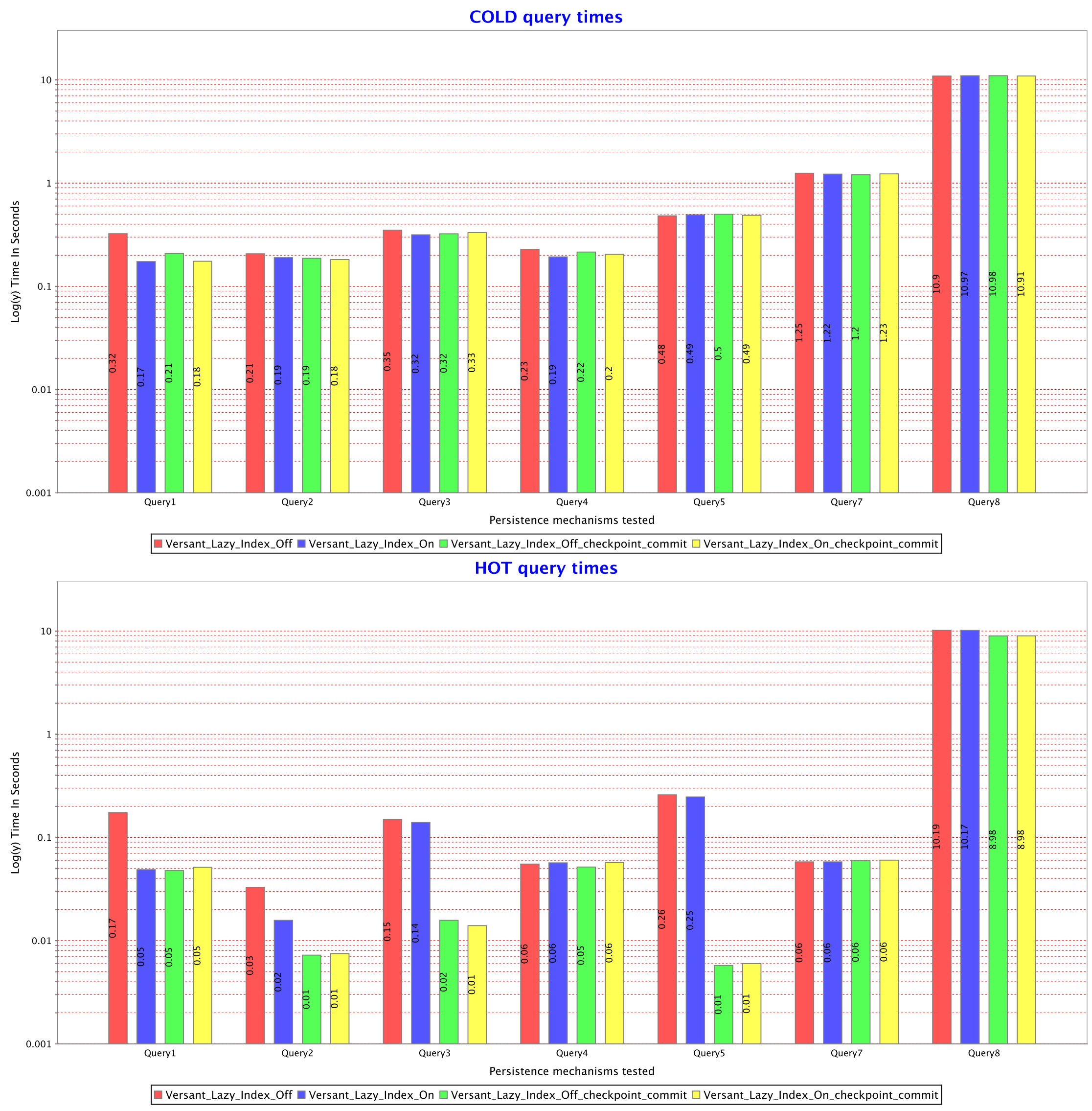


Figure 5.18: Versant cold and hot query results

Figure 5.19 shows that queries using `checkpointCommit()` have on average a 78% improvement due to the cache. Queries using `commit()` only have a 61% average improvement.

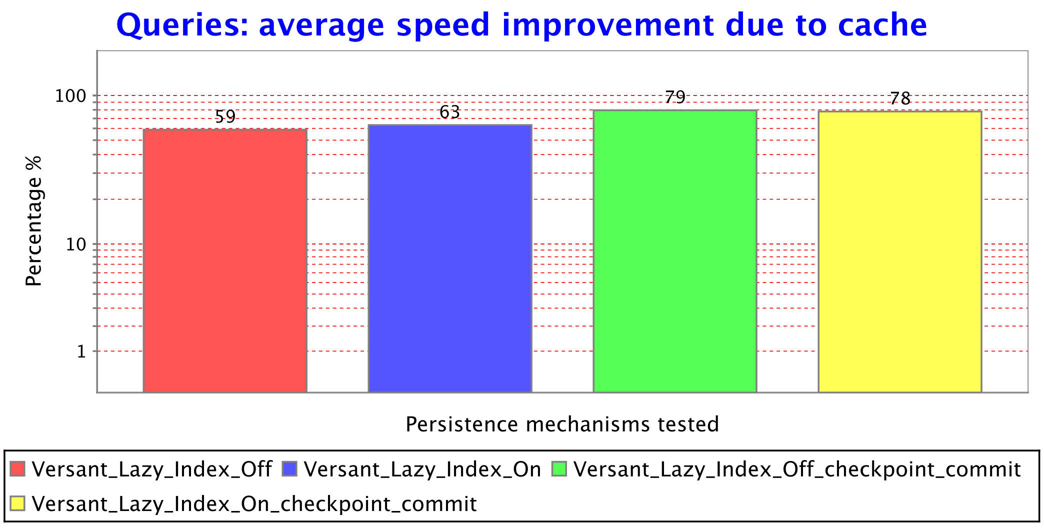


Figure 5.19: Versant average speed improvement due to cache for query runs

5.3.4 Modifications: Insert and delete

Insert and delete times are shown in Figure 5.20 and 5.3.4. The times for the *different configurations* are very close. It is noted that inserts are faster than deletes.

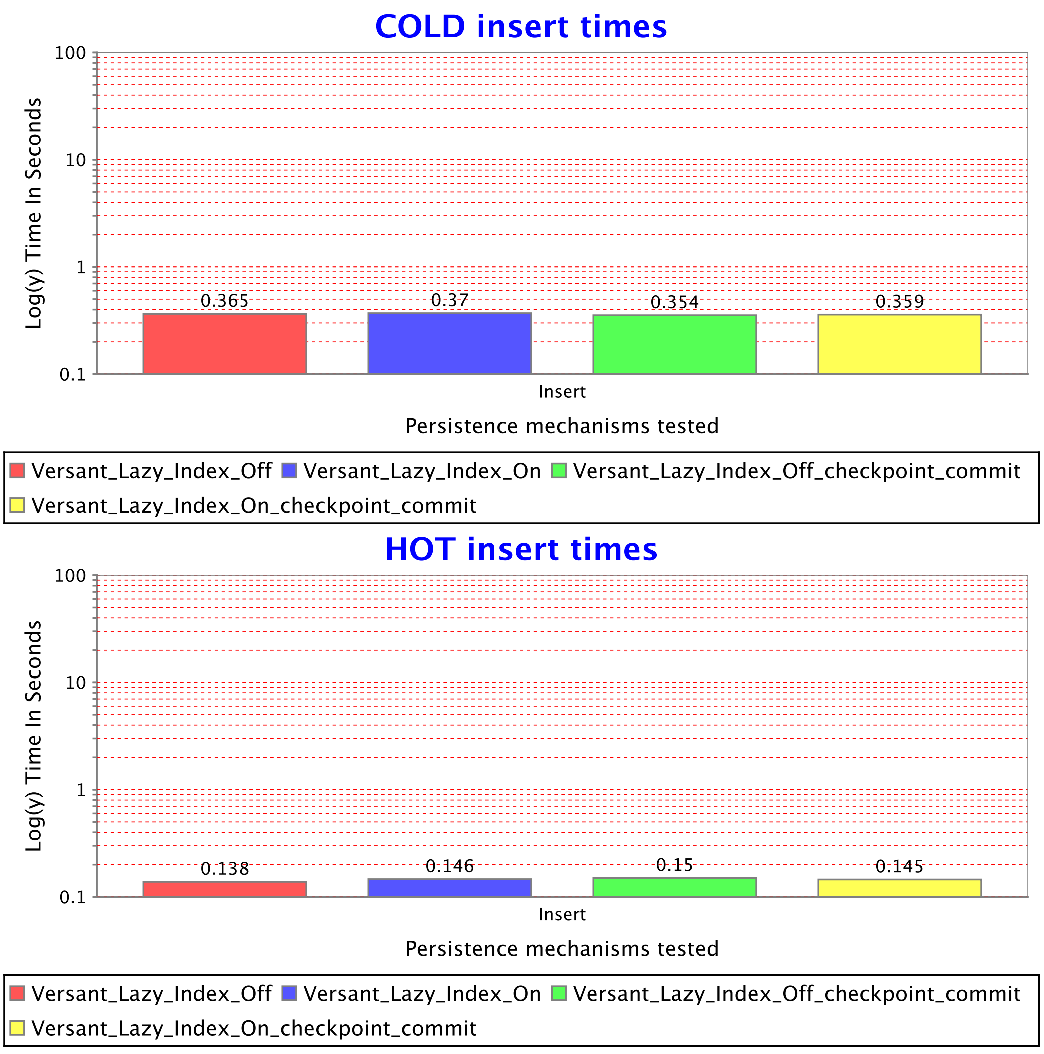
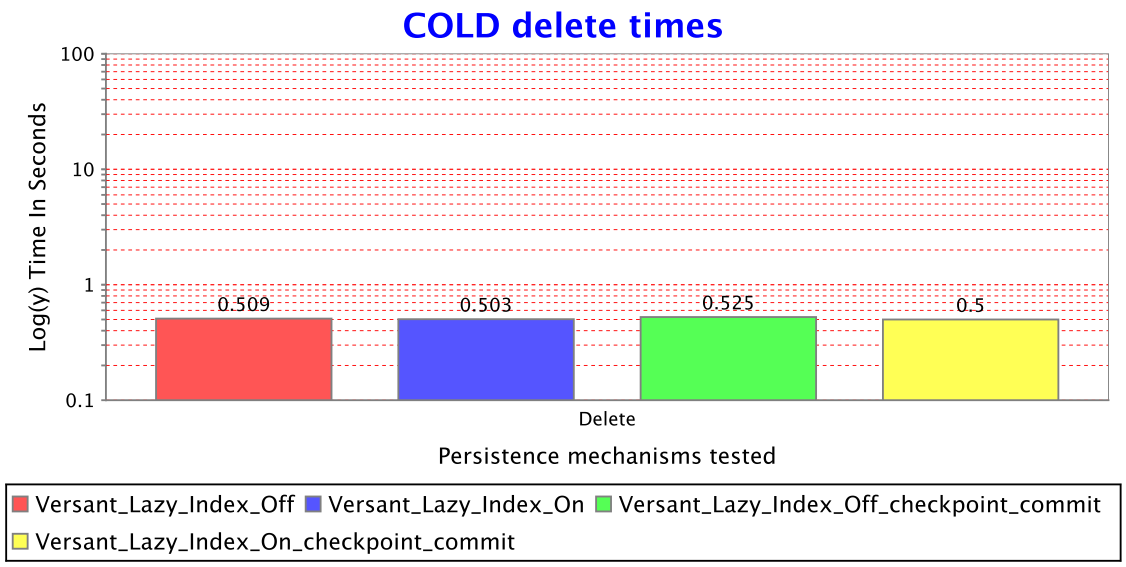


Figure 5.20: Versant cold and hot insert results



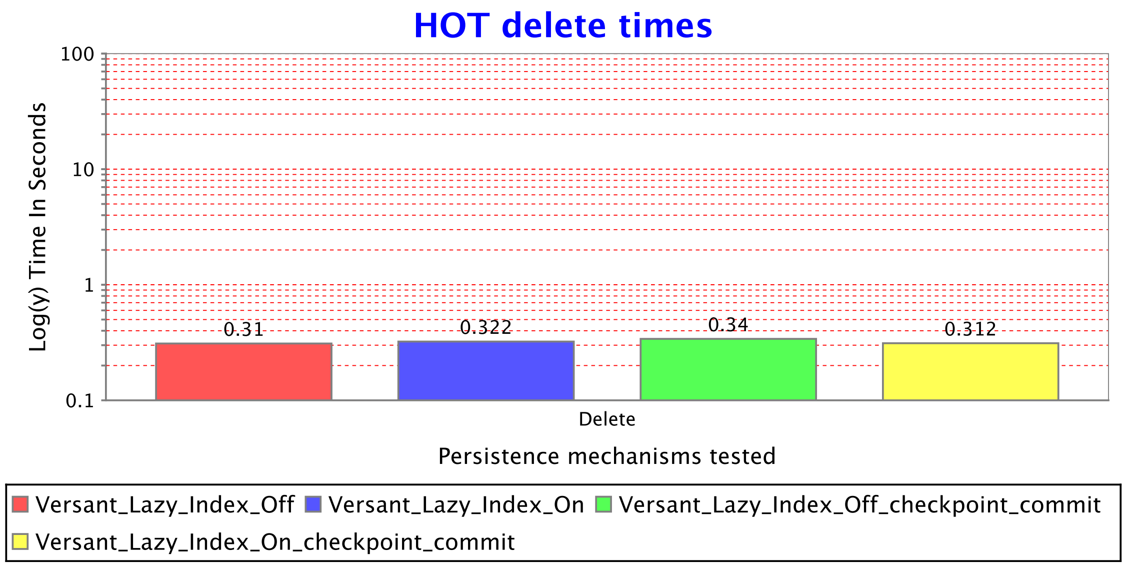


Figure 5.21: Versant cold and hot delete results

Like db4o when indexes are used there is almost no overhead. It was expected that when an index is used, during inserts and deletes, that these operations will be visibly slower. As mentioned in section 5.2.4 it seems that because a low number of objects are being inserted and deleted there is no major overhead for the index structure.

Table 5.3 shows speed improvements due to the cache. The cache assists inserts clearly more than deletes.

Configuration	% Improvement for Inserts	% Improvement for Deletes
Lazy loading and indexes off	62	39
Lazy loading and indexes on	60	36
Lazy loading with indexes off and using checkpoint commits	58	35
Lazy loading with indexes on and using checkpoint commits	60	38

Table 5.3: Versant average speed improvement due to cache for modification runs

5.4 Summary

In this chapter db4o, Versant and Hibernate were evaluated independently of each other. In the next chapter these products will be compared against each other.

Chapter 6

Comparisons and interpretations

In this chapter Hibernate, db4o and Versant are compared to each other. Only client-server configurations are compared. It is not fair to compare embedded mode with client-server. Future work could include comparisons of different embedded databases with each other.

For each object database and ORM tool, the configuration which gave the best overall result in the previous chapter is used in the comparisons.

The following configurations were selected:

- db4o's lazy and eager configurations using indexes. The eager run is included because for this small OO7 database eager loading improves the traversal times and is the fastest configuration of db4o for traversals. Only the networked client-server implementations for db4o are included. The *networked client-server mode* is selected because Hibernate and Versant also uses a client-server architecture.

- Hibernate's lazy configurations using indexes. Included are the configurations using Hibernate annotations as annotations seem to be the norm today. Hibernate XML configurations, which were quite fast, are included as well.

- Versant's configuration using its custom persistent collections, indexes and checkpoint commits. The checkpoint commits do not clear the associated session cache when a transaction is committed. Hibernate and db4o also do not clear their session caches after commits.

It should be emphasised that the results presented in this chapter have already been given in the previous chapter. It is only the presentation style that differs. In the previous chapter, results were presented per ODBMS/ORM tool, while here they are explicitly crossed-compared.

The chapter is divided into two sections. The first gives the actual cross-comparative data, and offers a few plausible explanations of the results. It should be emphasised that such explanations are in the nature of tentative hypotheses, and require further study to be confirmed. Such confirmation should be seen as being beyond the scope of this limited study. The second section relies on a rough scheme for scoring the overall performance of the respective tools. It is emphasised that the interpretation of the outcome of such scoring mechanisms needs to be approached rather cautiously.

6.1 Comparison results

In this section the results of the different implementations are compared and investigated. Again the results are grouped into creation, traversal, query and modification sections.

6.1.1 Creation

Figure 6.1 shows the creation times. Versant is clearly the fastest followed by Hibernate and then db4o.

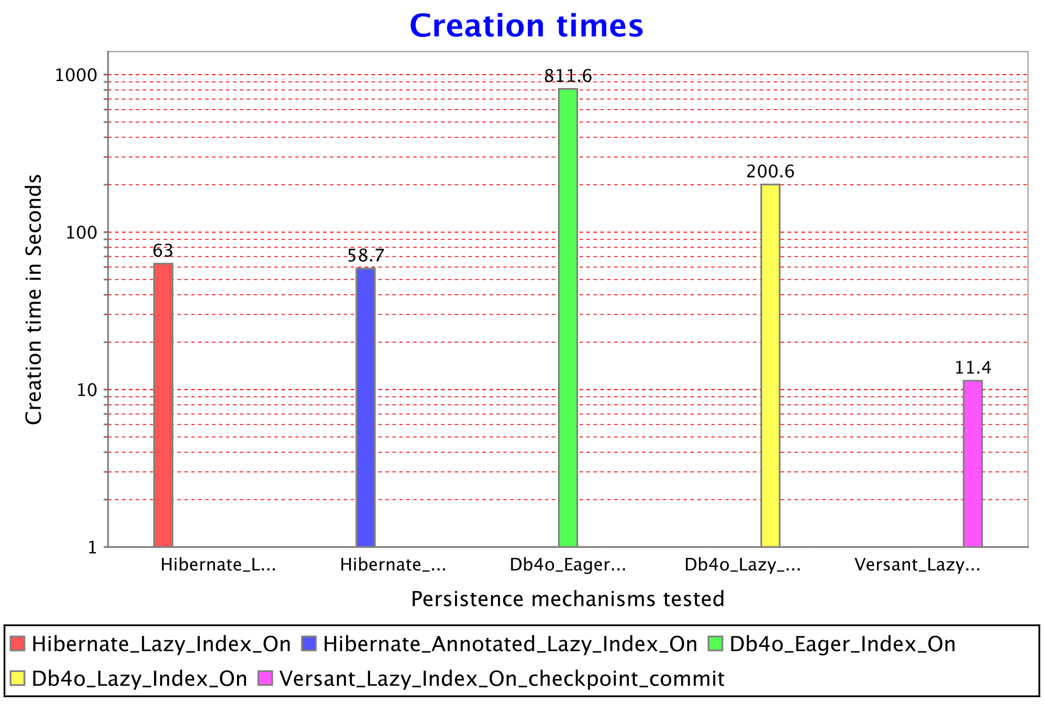


Figure 6.1: Creation times

Configuration	Database size
db4o with lazy loading and indexes on	Between 6 - 9 MB
db4o with eager loading and indexes on	Between 6 - 9 MB
Hibernate using mapping files with lazy loading and indexes on	± 71MB
Hibernate using annotations with lazy loading, indexes on and named queries	± 71MB
Versant with indexes on and using checkpoint commits	10 MB for database file and in total with system files 16MB

Table 6.1: Database sizes

The database sizes are shown in Table 6.1. db4o's OO7 database file size is the smallest with PostgreSQL being the largest at ± 71MB. db4o is an embedded database with a focus on creating small databases for embedded devices so it is not unexpected that it has the smallest database file.

[1]Taking a deeper look at the PostgreSQL database it was found that for example the `AtomicPart` table is 18MB with the indexes on that table alone being 11MB. The `AtomicPart` table contains indexes on two `AtomicPart` fields and have the largest index table of all the tables. The `Connection` table, with its 30 000 connections, has a size of 30MB.

6.1.2 Traversals

Versant is the fastest for most of the *cold traversal runs* in Figure 6.2 except for *Trav8* and *Trav9* where Hibernate with annotations are slightly quicker. Hibernate's time for *Trav8* and *Trav9* are 0.14 and 0.13 seconds respectively versus Versant's 0.16 and 0.16 seconds.

In the *hot traversal runs* Versant again is the fastest except for *Trav8* and *Trav9*. Hibernate and db4o's eager loading configuration are faster in these traversals. They complete in an insignificant amount of time. For *Trav9* and *Trav8* some result bars in the figures appear to be missing. This is because the values are so close to zero that they are not visible on the current scale.

Trav8 and *Trav9* merely scans the `Models, Manual` object for string

[1]pgAdmin 3 was used to investigate the table sizes. To find the total size the following command was used: select pg_size_pretty(pg_database_size('oo7jsmall')) ;

matches. Since they find and operate on one object, it is in line with expectation that they are very fast.

The db4o eager loading configuration is also slightly faster, by 0.01 second, than Versant for *Trav1* in the hot run.

Hibernate, using the XML configuration, is the slowest in the following cold traversals: *Trav1, Trav2a, Trav2b, Trav3a* and *Trav3b.* Hibernate with annotations and named queries is faster than db4o in *Trav2c* and *Trav3c* where a large number of objects are traversed and modified and where the cache plays an important role.

In the hot runs and most of the cold runs, db4o's eager loading configuration performs better than its lazy loading version because of the caching of all the data in the first run. The db4o eager loading configuration also has less communication with the db4o server, as the whole tree is cached on the first run.

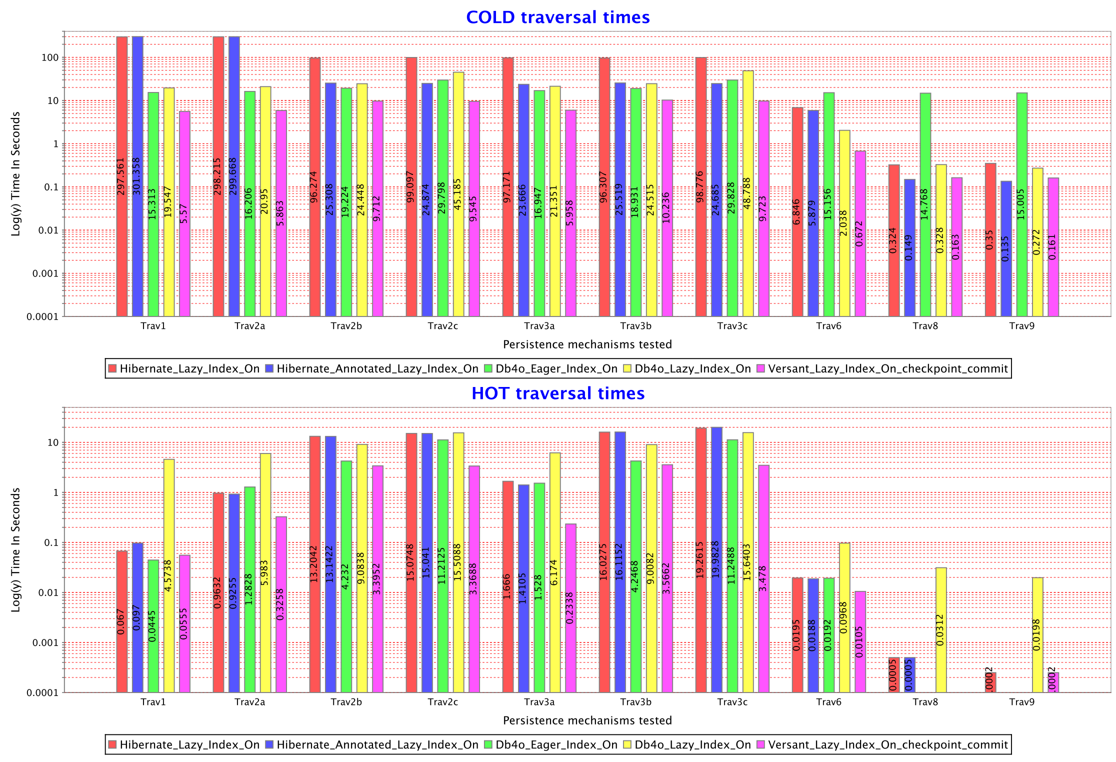


Figure 6.2: Comparison results of traversals

Taking a further look at some of the traversals the following is observed:

- Cold runs: For *Trav1*, both db4o's lazy configuration and Hibernate's lazy configurations are slower than the rest. Also note that *Trav1 and*

Trav2b traverse the *same number* of objects but the objects are modified in *Trav2b*. When objects are changed in *Trav2b* the Hibernate times actually improve compared to the Hibernate times for *Trav1*. For *Trav2b* Hibernate's times improve from around 300 seconds down to 90 and 25 seconds while the db4o and Versant configurations are ± 4 seconds slower. Versant and db4o are slower because more work is performed as the objects are being modified as well. It would seem that Hibernate with PostgreSQL times improve when modifications are made. It could be that the changes are not flushed immediately. db4o and Hibernates times are very slow for *Trav1 (*traverse 43 700 objects*)* and *Trav2b* compared to Versant. db4o's times are between 25 and 15 seconds and Hibernate is between 300 and 25 seconds compared to Versant's 5 and 9 seconds. *It is clear, that when many objects are traversed and modified in the cold Trav2b run, that Versant scales better.*

- Hot runs: Looking at *Trav1* and *Trav2b* times in the hot runs, it is observed that the times now increase for all configurations when running and modifying objects in *Trav2b* compared to their times when merely traversing the objects in *Trav1*. Although *Trav1* times are very close, db4o's eager configuration is slightly faster by 0.01 second from Versant. Moving from *Trav1* to *Trav2b* Hibernate's times increase by ± 13 seconds.

- Hot runs: In *Trav2a* and *Trav3a*, which involve changes to a small number of objects (2185), Versant and Hibernate's annotation configuration are faster than db4o.

- Hot and cold runs: In larger traversals like *Trav2c* and *Trav3c*, Versant is the fastest. If once-off traversals are needed, that involve large number of objects, Versant performs better.

- Hot and cold runs: Traversal 3 is similar to Traversal 2 but, as stated previously, involves changing or updating an *index field*. The difference between traversal *Trav2b+Trav3b* (traverse 43 700 objects) and *Trav2c+Trav3c* (traverse 174800 objects) is that the *"c" traversals* update each `AtomicPart`, in a `CompositePart` *four* times *per iteration*, while the *"b" traversals* only carry out the update *once per iteration*. Thus performing traversals *Trav3c* and *Trav3b*, in a hot runs, the db4o cache performs better than Hibernate on multiple immediate repeats. In the cold runs for *Trav3c* the Hibernate annotation configuration with named queries performs slightly better. Traversals *Trav2b* and *Trav3b* changes an index field which could create an overhead as the index is being updated. While the overhead of changing the index field is also present in *Trav3c*, it is still very fast because there are a larger number of objects involved that are more effectively cached. The Hibernate cache assists so that the Hibernate times only increase by 3 seconds when performing *Trav2b* and *Trav3b* in the hot runs. The db4o times increase by ± 6 seconds when performing *Trav2b* and *Trav3b* in the hot runs. It would be interesting to know if a hash index or b-tree index is used by Hibernate and to see the performance difference between the two if any.

- Hot and cold run: Carey et al. [45] mention that times increase between *Trav2a* and *Trav2*b because the number of objects being traversed and updated increase. The same is true for *Trav*3b and *Trav*3c. Carey et al. [45] further states that for Exodus (an object database included in their study), moving from *Trav2*b to *Trav2*c should not increase the times because there are "repeated updates to a cached object" and that for Exodus "only the last change is logged". For db4o the traversals times increase moving from *Trav2*b to *Trav2*c. The db4o eager loading configuration time increases from 4 seconds for *Trav2*b to 11 seconds for *Trav2*c in the hot run. This could be related to how db4o transaction and commits are implemented. The db4o reference manual [62] states that "Objects created or instantiated within one db4o transaction are written to a temporary transaction area in the database file and are only durable after the transaction is committed". This writing to "a temporary transaction area" is also done for existing objects that are updated. Note that changes are only made durable after commit. Thus, each time an object is changed and saved during *Trav2*c, a call (or message as it is called in db4o) to the server takes place. In db4o this *default* behaviour could be changed by batching calls using the `clientServer().batchMessages(true)` setting [62, page 915]. It was later found that according to the Java Doc comments in db4o 7.4, the default has been changed and messages are now batched. However even though messages are batched, there is still a overhead in the db4o case. This overhead could be related to the way in which messages are batched or the batching of messages is implemented in db4o. Versant only updates the database once commit is called. Hibernate also only synchronises changes to objects in the Session to the database when commit is called or an explicit flush is called [39, p 183][2]. Figure 6.2

[2]Bauer and King state that: "The Hibernate Session implements transparent write behind. Changes to the domain model made in the scope of a Session aren't immediately propagated to the database. This allows Hibernate to coalesce many changes into a

shows that Hibernate's and Versant's times remain consistent between *Trav2*b to *Trav2*c.

- In all the hot runs db4o's lazy loading or manual lazy activation seems to create an overhead compared to the eager loading configuration. These db4o lazy runs are the slowest. For example in *Trav2a* the db4o eager loading configuration runs on average ±1 seconds while the db4o lazy configuration takes ±5 seconds.

- Cold run: For *Trav6* Versant is the fastest at 0.6 seconds. db4o lazy loading configurations are second at ±2 seconds followed by Hibernate which is between 7 and 5 seconds. *Trav6* traverses the same number of objects as in *Trav2a* and *Trav3a* but with no updates to any fields.

- Hot run: For *Trav6* Versant is slightly faster than Hibernate followed by db4o. The cache helps to improve all the configuration times and bring them closer together.

Looking at the speed improvement due to cache use in Figure 6.3 it is clear that the Hibernate XML mapping implementation has the best improvement (around 93%). This cache improvement helps Hibernate to be faster than db4o in *some* of the hot runs. This improvement is due to Hibernate's *first level cache.*

Versant's cache provides a 85% improvement while the db4o cache provides on average a 80% speed improvement. In summary *Versant performs the best overall* for traversals.

In the next section the query performance is investigated.

minimal number of database requests, helping minimise the impact of network latency."

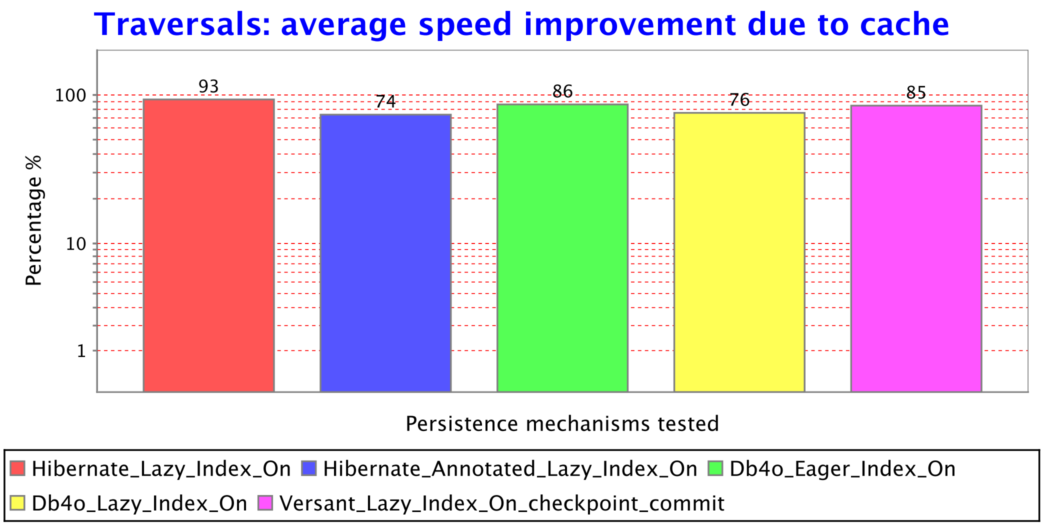


Figure 6.3: Comparing traversal and query average speed improvement due to cache for all

6.1.3 Queries

The query results are shown in Figure 6.4. *Versant is the fastest for all the cold and hot query runs.* The only exception is for the extra query that was added for Hibernate, Query8Join. Using this join query, Hibernate is the fastest for Query 8.

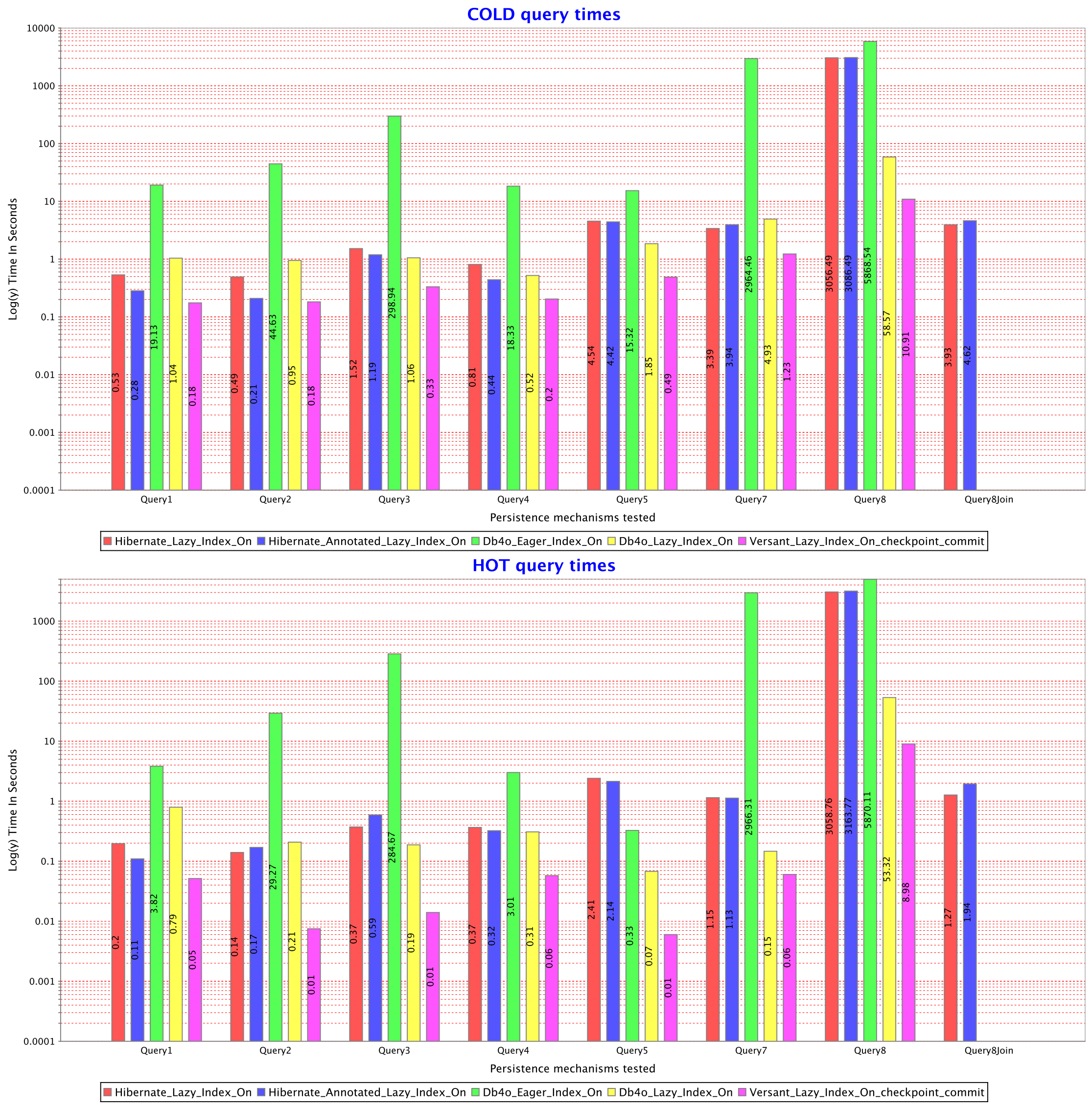


Figure 6.4: Comparison results of queries

For the query runs db4o eager configuration (green bar) is slower than the db4o lazy configuration (yellow bar). This is a reversal of the traversal results in which the db4o eager configuration was faster than the db4o lazy configuration.

Query 8 is the most interesting query as it involves a join between `AtomicPart`'s and `Document` objects. As discussed in the previous chapter, Query 8 suffers from the *n+1 select* problem meaning that there are extra query calls to the server and that if eager loading is used extra collections are retrieved that are not used. Versant performs Query 8 in 10 seconds in the cold run while its *average hot run time* is 9 seconds.

Hibernate runs Query 8, which suffers from the *n+1 select* problem, in ± 3050 seconds and 3058 seconds respectively in the cold run and average hot run cases. When Hibernate's Query 8 is rewritten using a join and only one request to the database, its times improves. Query8Join times are ±4 seconds in the cold run, and its *average hot run time* is ±1.5 seconds.

db4o and Versant do not provide a join-like query. Times for Versant and db4o could be improved by using different techniques like batching query calls, or creating look-up collections (such as views in SQL). These techniques could improve the object database query times but the main focus was on using *query language* facilities.

Taking a look at the rest of query runs it is observed that:

- Query 1 is an exact match query. It involves finding `AtomicPart` objects by matching on an "ID" field on which there is an index. Hibernate and Versant perform this query on an index field faster than db4o in the hot and cold runs. db4o using eager loading perform Query 1 in ±19 seconds and db4o using lazy loading takes around 1 second. Versant and Hibernate perform Query 1 in less that 0.5 seconds.

- Query 2 and Query 3 are range queries. Query 2 finds 1% of the `AtomicPart`'s and Query 3 finds 10% of the `AtomicPart`'s in a certain range. It is expected that Query 3 will be slower than Query 2 as it finds more objects. Figure 6.4 shows that this is indeed the case. db4o's lazy configuration times are slightly faster than Hibernate in Query 3 in the hot run, with this increase in the number of objects involved in the range. db4o performs Query 3 in ± 0.2 seconds while the Hibernate configurations takes between 0.4 and 0.6 seconds. However, Carey et al. [46] state that Query 2 and Query 3 are "candidates for b+tree lookups". db4o does indeed use a b-tree index to improve query speeds. It would be interesting to change some of the settings provided by db4o for this b-tree and find the performance impact that they might have on these two queries. These settings are: the `bTreeCacheHeight(height)` which "configures the size of BTree nodes in indexes (sic)" and the `bTreeNodeSize(size)` which "configures caching of b-tree nodes" [62]. Interesting to note is that "clean b-tree nodes will be unloaded on commit and rollback unless they are configured as cached here" [62]. This cleaning could influence the performance as well. Hibernate increases by almost 1 second between Query 2 and Query 3 in the cold run while db4o and Versant increases by ± 0.15 seconds.

- Query 5 is similar to Query 8 in that it suffers from the *n+1 select* problem. Query 5 first selects all the `BaseAssembly`'s and then visits each one. Each visit requires that the private components (`CompositePart`'s) are looked up and it, too, is then traversed. While visiting each `CompositePart`'s, its fields are being compared. It is clear that Query 5 also suffers from the *n+1 select* problem. In these situations the object databases perform better. If this query is rewritten using a join and the comparisons are done once on the server, at the time the join is executed, the times for Hibernate should improve. Also note that in Query 5, db4o's eager configuration times improve (from 15 to 0.3 seconds) in the hot run. This is because of the fact that objects and their associated collections are retrieved and cached throughout the iterations. This is the only scenario where the eager loading configuration of db4o is faster than Hibernate. Query 5 is more a traversal in nature than a query.

- Query 7 is also queries 10 000 `AtomicPart` but is slightly faster than Query 8 because no *joins* or *n+1* queries are involved. Hibernate and Versant are faster than db4o for Query 7 in the cold runs. db4o's lazy configuration improves in the hot run to be ahead of the Hibernate configurations. Query 7 does not involve joins or accessing collections and is just selecting objects of *one type/class* and iterating through them. For Hibernate this means it accesses one table.

Overall, Hibernate is mostly faster than db4o in the following queries: Query 1, Query 2, Query 4, Query 7 and in Query 8 using the the *join* query*(*Query8Join). Thus Hibernate is faster than db4o in 5 out of the 7 queries in the cold runs.

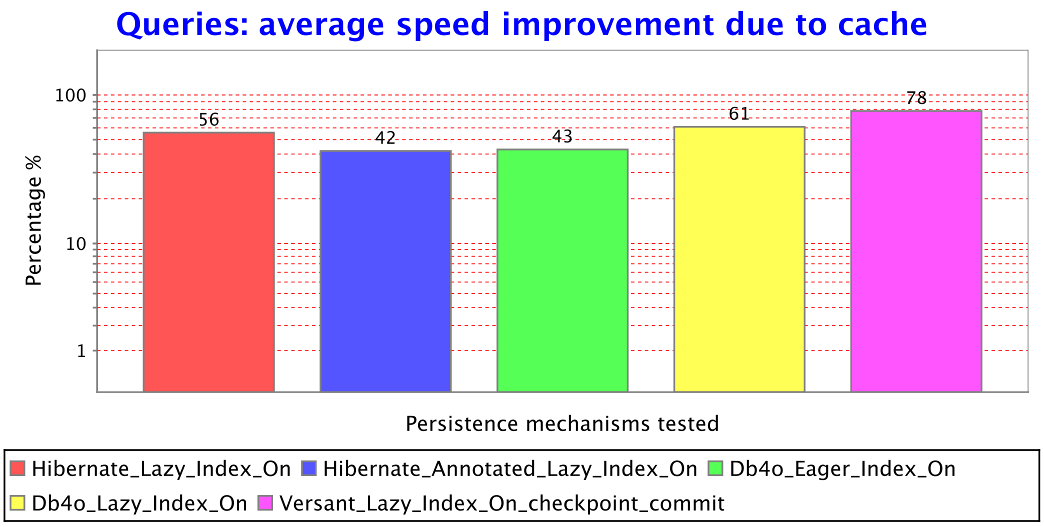


Figure 6.5: Comparing traversal and query average speed improvement due to cache for all

Figure 6.5 shows the speed improvements due to cache utilisation. Versant's query speed is clearly assisted by a cache while db4o and Hibernate's caches appear to assist with *traversals more than with queries*. Versant's cache assists with both traversals and queries [98].

It would seem that db4o and Hibernate caches either do not cache query results automatically, or if they do so, then they do not do so as well as Versant. In Hibernate an extra query cache is provided but it is not on by default. Future work should include turning on the Hibernate query cache. This should increase the Hibernate performance.

Note, finally, that it is important to know what type the query is and how many requests are sent on to the server.

6.1.4 Modifications: Insert and delete

Figure 6.6 shows that Versant is the fastest for cold and hot inserts, followed by the db4o lazy configuration. Versant is also the fastest for cold and hot deletes, followed by db4o.

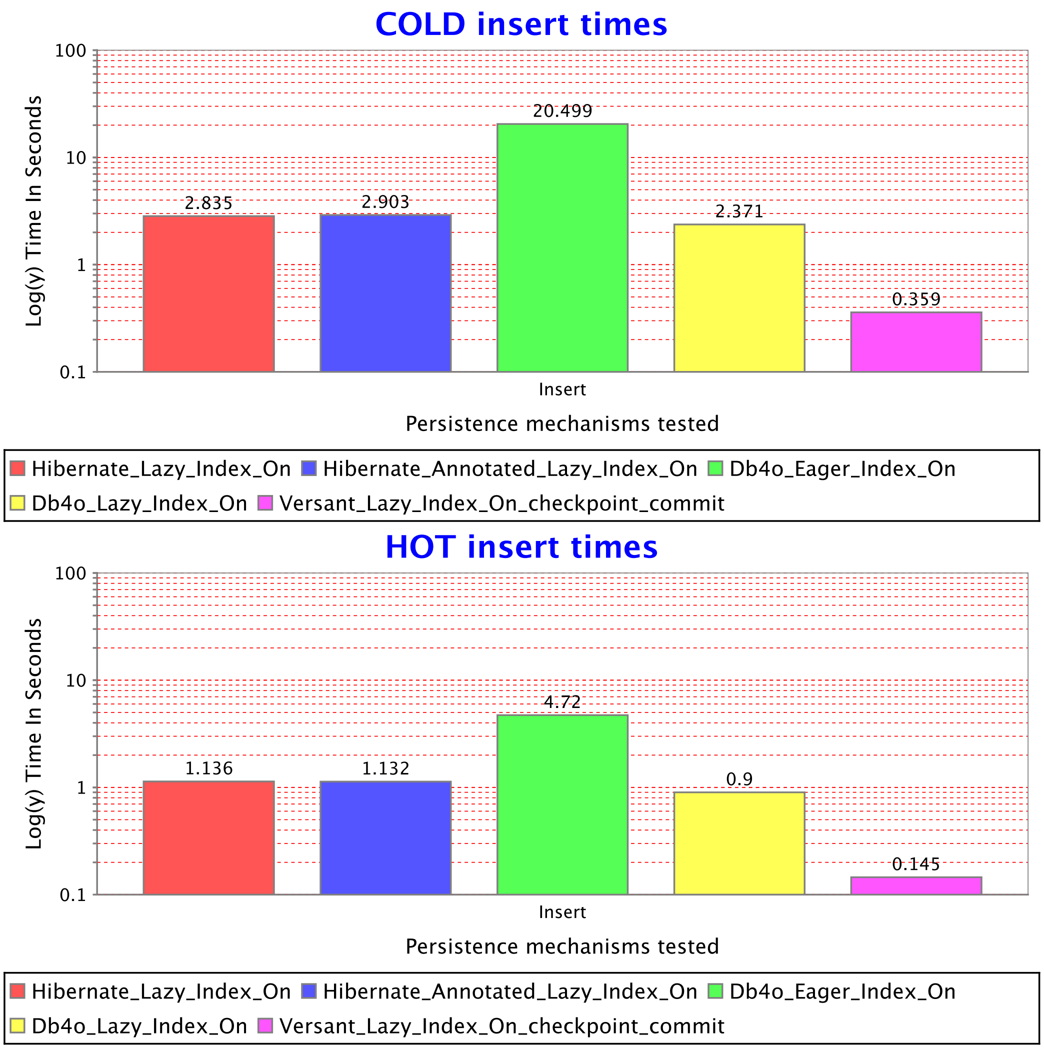


Figure 6.6: Comparison result of inserts

The db4o eager configuration is the slowest because it unnecessarily retrieves extra objects into memory during insertion and deletion.

For db4o and Hibernate it seems that deletes are faster than inserts. Versant is faster performing inserts and slightly slower when performing deletes.

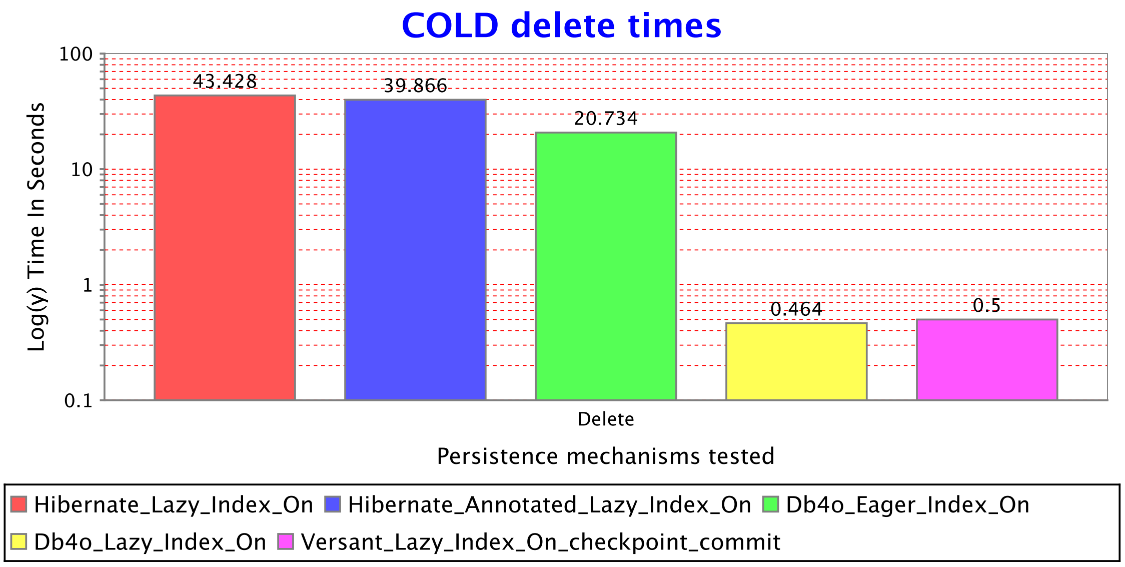
db4o provides a *cascade on delete* property that can be set on a class or on a field of a class. Hibernate also provides an *cascade on delete* property that can be set on associations. However, Versant does not provide such a delete option.

The OO7 delete operation definition is as follows:

"Delete the five newly created composite parts (and all of their associated atomic parts and document objects)"

This means that when a `CompositePart` is deleted, its associated `AtomicPart` and `Document` objects that were added during the insert are also deleted. Thus the database should be taken back to the same state as before the inserts.

As a result, the database should contain the same number of objects as before the inserts. Furthermore, when deleting an `AtomicPart` its associated `Connection` objects should also be deleted.



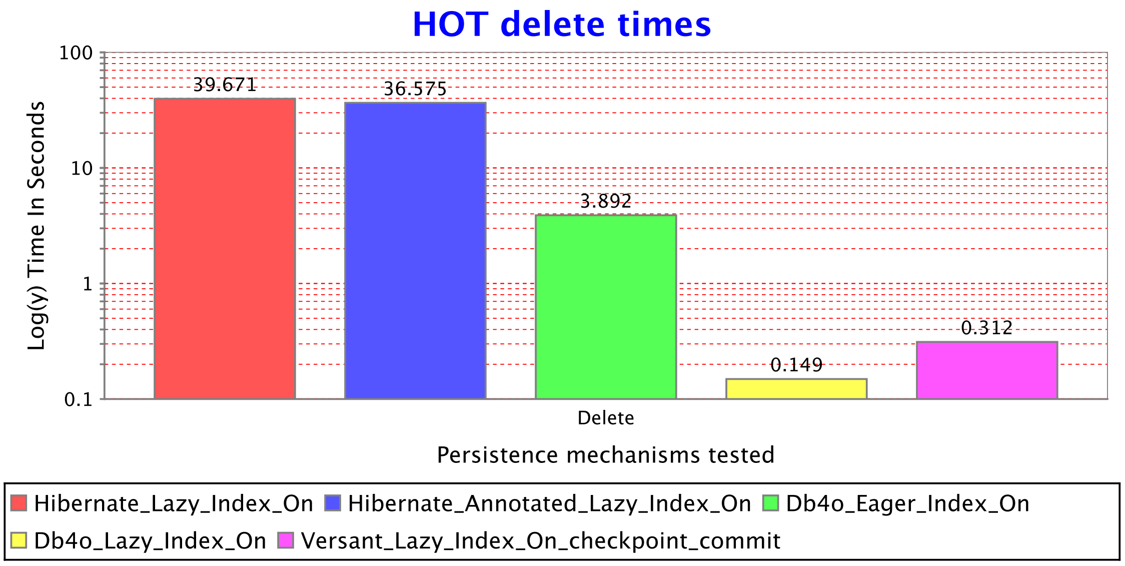


Figure 6.7: Comparison result of deletes

In OO7, the delete operation first iterates through each `CompositePart` object and then access each associated object, marking these objects for deletion. The `CompositePart` objects had been linked to `BaseAssembly` objects during insertion by adding them to `BaseAssembly` collections. In the deletion task, these `CompositePart` objects are first removed from their collections and then marked for deletion. Their `Document` and `AtomicPart` objects are also marked for deletion.

When an `AtomicPart` is deleted in db4o or Hibernate, the deletion is cascaded to their associated `Connection` objects.

Because Versant does not provide a *cascade on delete* option, each of the `Connection` objects needed to be deleted explicitly, .i.e. to delete a `Connection` and its associated objects, a manual method was called. This method iterates through all `Connection` objects, marking them for deletion.

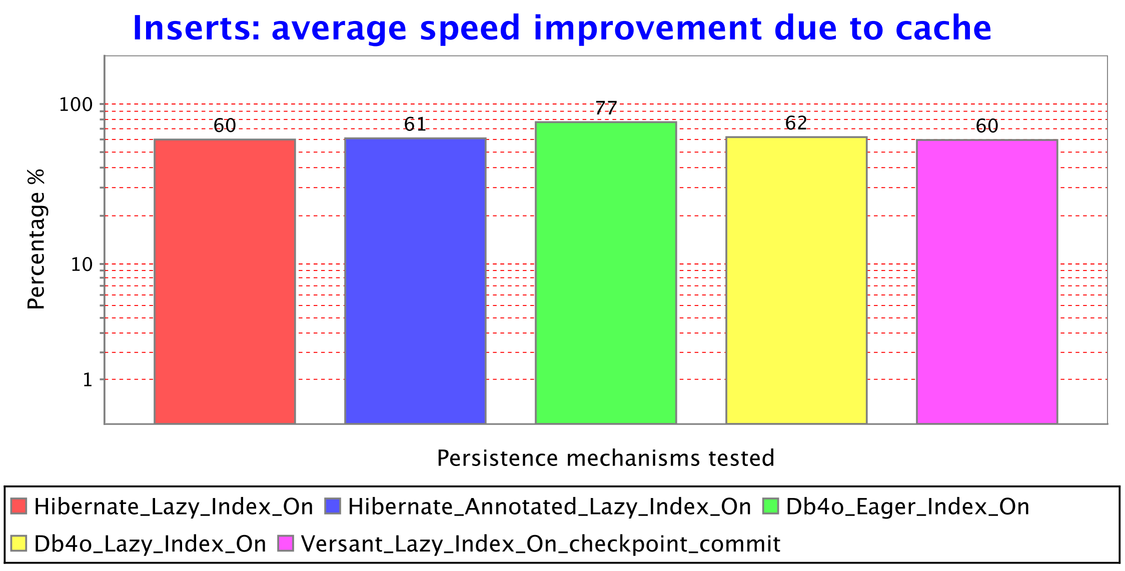
This was done, even though Versant provides a hook (or callback) called vDelete [100] which can be invoked when an object is to be deleted. This allows a user-defined action to be called to delete the associated objects and collections. vDelete was not used as there wasn't enough information available on how to implement it, plus it seems that it only provides a automatic way to call the *manual method* which is already implemented.

Versant performs deletions faster than db4o and Hibernate which uses the *cascade on delete* facility.

Hibernate is very slow to delete these objects. Its slowness could be related to the size of the tables. The `Connection` table is $\pm$ 30MB.

During insertion, random `BaseAssembly` objects are looked-up, using their ids, and `CompositeParts` are added to their collections. Deletion also involve looking up the `CompositeParts` using their ids. Because Versant is quite fast for queries (look-ups) it could explain the edge that it has over db4o and Hibernate for inserts and deletes in the OO7 benchmark.

The speed improvement due to the cache can be seen in Figure 6.8. On average Hibernate's speed improvement due to the cache, for inserts, is 60%, db4o 70% and Versant at 60%. The cache assists less for deletes. The only exception is db4o. The db4o cache seems to assist during deletes. On average Hibernate's speed improvement due to the cache, for deletes, is 8%, db4o 74% and Versant at 38%.



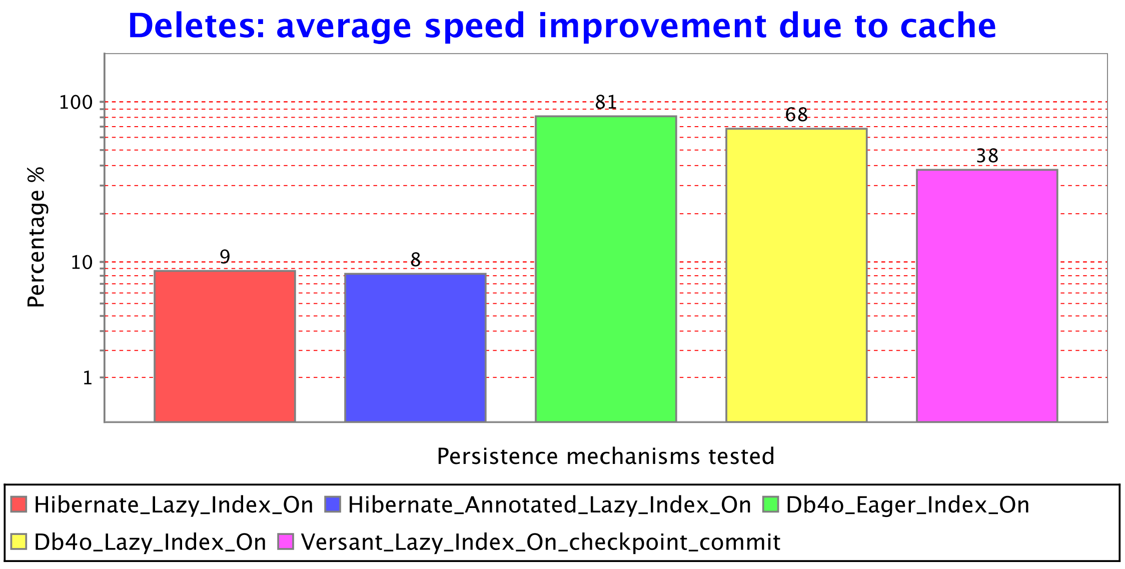


Figure 6.8: Comparing insert and delete average speed improvement due to cache for all

6.2 Scoring the benchmark results

In view of the results given in the previous section, one is tempted to ask which of the tools is the overall winner. While the earlier benchmarks used single value results and rankings that would seemingly answer this question, like TPC-A [52], the OO7 benchmark provides a collection of results.

To determine an overall winner Carey et al. point out that a few approaches can be used to create a "single number that can then be used to rank systems" [49, 47]. Nevertheless, they and Cattell and Skeen[52] warn against single number results and ranking, pointing out that they may be "narrow", since valuable information may be lost when looking at a ranking. In fact Carey et al. demonstrate how different ranking approaches can provide different results.

While noting the cautions of Carey et al, two of these approaches are briefly explored below: the number of first places on the benchmark and the weighted ranking of places. These are discussed in the following sub-sections.

6.2.1 Number of first places

The number of first places on the benchmark results are provided in Table 6.2. This ranking is created by counting the number of first places for each object database and ORM tool tested, taken over all the OO7 benchmark operations [47]. This ranking system does not provide a clear picture and should not be used to make a decision. The *weighted ranking of places* provide a better picture of the performance of the systems.

In the previous chapters it was shown that Hibernate performs better with join like queries and that Query 8 performance suffers from the *n+1 select problem.* For this reason, in calculating these scores, the new Hibernate *Query8Join* using the the join query was used instead of the normal Query 8. The study uses the best available time for each product in the ranking calculations.

Rank	Persistence Mechanism/Object database/ ORM Tool	Score
1	Versant	29
2	Hibernate using annotations with lazy loading, indexes on and named queries	3
3	db4o with eager loading and indexes on	2
4	db4o with lazy loading and indexes on	2
5	Hibernate using mapping files with lazy loading and indexes on	2

Table 6.2: Ranking according to the number of first places

6.2.2 Weighted ranking

The weighted ranking systems works as follows: Each system gets "one point for a first place finish on a test, two points for a second place finish, three points for a third place place", etc [47]. These points are added together and the system with the lowest number is the best. These weighted rankings are shown in Table 6.3. Overall the db4o eager loading configuration is now slower than the db4o lazy loading configuration.

Rank	Persistence Mechanism/Object database/ ORM Tool	Score
1	Versant	51
2	Hibernate using annotations with lazy loading, indexes on and named queries	116
3	db4o with lazy loading and indexes on	125
4	db4o with eager loading and indexes on	137
5	Hibernate using mapping files with lazy loading and indexes on	141

Table 6.3: Ranking according to weighted ranking of places

The foregoing assessment is rather course-grained. As an alternative, the weighted ranking results may be computed at a finer level of granularity. Consequently, weighted ranking results have been computed separately for the cold and hot runs. Additionally, ranking results have also been broken down separately for traversals and queries. These results are shown in the following subsections.

6.2.2.1 Weighted ranking for cold runs

If the application under consideration operates mostly on data that is not repeated and mostly on disk, then the cold run ranking is very important. The weighted ranking results for just cold runs are shown in Table 6.4 below. The Hibernate configuration using mapping files has now moved up a place to 4th place.

Rank	Persistence Mechanism/Object database/ ORM Tool	Score
1	Versant	24
2	Hibernate using annotations with lazy loading, indexes on and named queries	56
3	db4o with lazy loading and indexes on	58
4	Hibernate using mapping files with lazy loading and indexes on	73
5	db4o with eager loading and indexes on	74

Table 6.4: Ranking according to weighted ranking of places for just the cold run

6.2.2.2 Weighted ranking for hot runs

If the application under consideration operates mostly on data that is *repeatedly accessed* and mostly *in-memory* then the hot run ranking is very important. The weighted ranking results for just hot runs are shown in Table 6.5 below. Versant is first followed by the Hibernate configuration using annotations. The db4o configurations are just behind the Hibernate configuration. The db4o eager loading configuration is better than the db4o lazy configuration overall in hot runs.

Rank	Persistence Mechanism/Object database/ ORM Tool	Score
1	Versant	27
2	Hibernate using annotations with lazy loading, indexes on and named queries	60
3	db4o with eager loading and indexes on	63
4	db4o with lazy loading and indexes on	67
5	Hibernate using mapping files with lazy loading and indexes on	68

Table 6.5: Ranking according to weighted ranking of places for just the hot run

6.2.2.3 Weighted ranking for cold query runs

If the application under consideration operates mostly on data that is *not repeated*, mostly *on disk* and involves queries, then the weighted ranking results based only on cold query runs in Table 6.6 below are relevant. The Hibernate configurations are just ahead of the db4o lazy configuration.

Rank	Persistence Mechanism/Object database/ ORM Tool	Score
1	Versant	9
2	Hibernate using annotations with lazy loading, indexes on and named queries	17
3	Hibernate using mapping files with lazy loading and indexes on	21
4	db4o with lazy loading and indexes on	23
5	db4o with eager loading and indexes on	35

Table 6.6: Ranking according to weighted ranking of places for just the cold query runs

6.2.2.4 Weighted ranking for hot query runs

If the application under consideration operates mostly on data that is *repeatedly accessed*, is mostly *in-memory* and involves queries then the weighted ranking results limited to hot query runs in Table 6.7 below are of interest. Now the db4o lazy configuration is ahead of the Hibernate configurations—a reversal from the cold runs

Rank	Persistence Mechanism/Object database/ ORM Tool	Score
1	Versant	9
2	db4o with lazy loading and indexes on	20
3	Hibernate using annotations with lazy loading, indexes on and named queries	21
4	Hibernate using mapping files with lazy loading and indexes on	22
5	db4o with eager loading and indexes on	33

Table 6.7: Ranking according to weighted ranking of places for just the hot query runs

6.2.2.5 Weighted ranking for cold traversal runs

If the application under consideration operates mostly on data that is *not repeated*, mostly *on disk* and involves traversals then the weighted ranking results, for just cold traversal runs in Table 6.8 below are relevant. The db4o eager configuration and the Hibernate annotation configuration are tied in 2nd place. The db4o lazy configuration is in 3rd place.

Rank	Persistence Mechanism/Object database/ ORM Tool	Score
1	Versant	12
2	Hibernate using annotations with lazy loading, indexes on and named queries	31
3	db4o with eager loading and indexes on	31
4	db4o with lazy loading and indexes on	32
5	Hibernate using mapping files with lazy loading and indexes on	44

Table 6.8: Ranking according to weighted ranking of places for just the cold traversal runs

6.2.2.6 Weighted ranking for hot traversal runs

Finally, if the application under consideration operates mostly on data that is *repeatedly accessed*, is mostly *in-memory* and involves traversals, then the weighted ranking results limited to hot traversal runs in Table 6.9 below should be considered. The db4o eager configuration is now in 2nd place ahead of the Hibernate configuration using annotations. The Hibernate configuration using XML have moved up to 4th place.

Rank	Persistence Mechanism/Object database/ ORM Tool	Score
1	Versant	15
2	db4o with eager loading and indexes on	22
3	Hibernate using annotations with lazy loading, indexes on and named queries	32
4	Hibernate using mapping files with lazy loading and indexes on	37
5	db4o with lazy loading and indexes on	44

Table 6.9: Ranking according to weighted ranking of places for just the hot traversal runs

6.2.3 Summary

Considering all the ranking results, it is clearly important to look deeper and not merely pay attention to the top ranking. Indeed, even at the finer level of granularity computed above, ranking results do not reflect the fact that if the application under consideration is to perform queries similar to Query 8, then Hibernate is the fastest—despite the fact that Versant's ranking was the best for queries. Furthermore, if the application does pointer traversals over cached data, with no modifications during the traversal, then db4o's eager loading configuration is slightly faster. Again, this is not shown in the ranking results above. Clearly, therefore, it is possible to lose valuable information if just *a one number ranking* system is used.

Carey et al. [49, 47] state that the OO7 benchmark results should rather be *interpreted* by users in such a way that they look at results that are *relevant* or close to their *application workloads* or demands. In [49], they mention the following workloads:

- "If your workload does a lot of pointer-chasing over cached data, look at" *Trav1* in the *hot run* for the small OO7 database configuration.
  - In this study the eager loading configuration of db4o (0.04 seconds) is the fastest for *Trav1* in the hot run. Versant at 0.05 and

Hibernate at 0.06 seconds are very close. All of them are good candidates.

- "If your workload does sparse traversals over data that you expect to be disk resident much of the time, look at" *Trav6* in the cold run.
  - Versant is the fastest for *Trav6* (0.2 seconds) in the cold run followed by db4o's lazy loading configuration (2 seconds).
- "If you do a lot of exact match queries over cached data, look at" Query 1 in the hot run.
  - Versant is the fastest for Query 1 ($\pm$ 0.2 seconds) in the hot run followed by the Hibernate configurations ($\pm$0.3 and $\pm$0.5 seconds), the Hibernate annotation with named queries configuration being slightly faster than the Hibernate XML configurations.
- "If you do range queries over disk-resident data, look at q2b and/or q2c (sic)."
  - Since OO7 has no "q2b" and "q2c" queries, it is assumed they meant Query 2 and Query 3 in the cold run. Versant is the fastest for both Query 2 (0.01 seconds) and Query 3 (0.01 seconds). In second place is the Hibernate configurations with the Hibernate XML configuration being slightly faster at 0.14 and 0.37 seconds respectively.

This chapter presented cross-comparative data, and offered a few plausible explanations of the results. Schemes for scoring the overall performance of the respective tools were also presented. The next chapter will present the results when incorporating vendor recommendations.

Chapter 7

Vendor recommendations

Initially the study only focused on using the products in their *default modes* (out of the box) and use them as a *normal programmer* would. This initial out of the box state then formed a basis to which optimisation techniques were added. The study never intended to fine-tune their performance to the hilt using *all* the performance techniques and tricks possible. As an afterthought and as a matter of curiosity the results in the previous chapters were submitted for review to db4o, Hibernate and Versant. The vendors then provided some recommendations on how to improve the performance of their products and current configurations.

These recommendations were reviewed and it was decided to selectively create configurations to test some of them and to measure the improvements. The configurations based on the recommendations were mainly those used for the comparisons. There was only one exception. The Hibernate XML mappings were not used. It was left out because the fastest Hibernate configuration, which was the configuration using annotations and named queries, was selected and improved using the recommendations.

The Hibernate recommendations were implemented using annotations. This meant that a branch for the code had to be inserted and the annotations in the classes had to change to those recommended by Hibernate. These recommended annotations will be exported to an XML file in future, to form the basis of the Hibernate XML configurations in future work.

The configurations that were selected:

- db4o's lazy and eager configurations using indexes. The eager run is included because for this small OO7 database eager loading improves the traversal times and is the fastest configuration of db4o for traversals. Running in *networked mode.*

- Versant's configuration using its custom persistent collections, indexes and checkpoint commits

- Hibernates lazy annotation configurations using indexes.

Note that while the vendors reviewed the work and results and provided recommendations it does not mean that they endorsed the results.

The next sub-sections will report on and discuss the recommendations made for each product.

7.1 Hibernate recommendations

Hibernate provided the following recommendations:

- In the OO7 model `ComplexAssembly` and `BaseAssembly` objects inherit from `Assembly`. A table was created per child class (base_assembly and complex_assembly) and a main table for the parent class (assembly table). This is the *table per class* strategy [76], also called the "joined subclass strategy" in the Hibernate Annotations documentation [74]. Hibernate recommended that the *table per class hierarchy (single table)* strategy should be used instead of the *table per class (joined)* strategy. They stated that the *table per class* strategy is the "worst in terms of performance and memory usage". Because all objects are placed into one table a *discriminator* value or tag is used to identify which entries are `ComplexAssembly` or `BaseAssembly`. This recommendation was implemented.

- Each `Module` has an associated `Manual`. This is a one-to-one relationship. Hibernate recommended that the object should be embedded in the parent class using the `@Embedded` and `@Embeddable` annotations. This approach meant that there was no table for `Manual`. `Manual`'s are now stored with their associated `Module` in the `Module` table. This recommendation was implemented.

- In the original mappings created for the Java OO7 model the foreign keys were explicitly specified. Hibernate will automatically select the right foreign key to use. There is no need to specify the foreign key. Hibernate recommended that foreign key selection be left up to the Hibernate engine. The explicit foreign key specifications were removed following the Hibernate recommendation.

- It is important that one side of a many-to-many and one-to-many bidirectional relationship mapping have the tag `inverse = xxx` (`mappedBy = xxx`) tag set to *true*. During the review it was found that this tag was missing. The inverse is needed otherwise it creates an extra overhead when being updated [76, ch 19.5]. Hibernate recommended that this tag be added. Hibernate provides the following rule: "All bi-directional associations need one side as inverse. In a one-to-many association it has to be the many-side, in many-to-many association you can pick either side, there is no difference" [76, ch 1.3.6]. Following Hibernate's recommendation the `inverse = xxx` (`mappedBy = xxx`) tag was added.

- Hibernate recommended changing the relationship mapping between `AtomicPart` and `Connection` objects. Hibernate recommended that one-to-many mappings be used *from* `AtomicPart` to `Connection` and that
  `Connection` then use many-to-one mappings *back* to the "from" and "to" `AtomicPart`. The current mapping was using many-to-many mappings from the `AtomicPart` to `Connection`. While testing the use of one-to-many and many-to-one it was found that when traversing, the incorrect number of objects were retrieved. The problem related to the relationship mapping between `AtomicPart` and `Connection`. Two possible solutions were suggested:

  – Continue using many-to-many mappings. This approach is not recommended by Hibernate.

  – During a deeper investigation of the model and the schema definition provided in Carey et al.[46] it was found that from-connections in `AtomicPart` are not used in the OO7 benchmark. Only the to-connections are used to traverse from-`AtomicPart` to the to-`AtomicPart` The extra from-connection collection were removed from all the implementations (db4o, Hibernate and Versant). The mappings as recommended by Hibernate were then implemented. The removal of the extra collection did not have a significant impact on the benchmark results. Also the correct number of objects were retrieved during traversals. The removal could cause eager loading results to be a bit faster because there is no need to load this extra (unneeded) collection into memory.

- Hibernate recommended the use of batching during database creation. The creation of the OO7 database is a large once-off process where objects are inserted into the database. This operation is rightly classified as a *batch process*. Hibernate provides strategies to handle batch processing and especially batch inserts [76, ch 13.1]. The batch insert strategy suggests that the session be *flushed* and *cleared* for every few object that are created and inserted in the database. The Hibernate Java Doc states that "Flushing is the process of synchronising the underlying persistent store with persistable state held in memory". They also state that clearing: "Completely clear the session. Evict all loaded instances and cancel all pending saves, updates and deletions". The batch insert strategy as recommended by Hibernate was implemented.

- Hibernate also suggested that the POJO's be cleaned-up by removing all persistent and linking related code from the constructors. This suggestion was implemented and a new class `OO7CreateDatabase` was created. This new class contains all the persistent and linking related code taken from the POJO constructors.

- Hibernate recommended the use of `evict()` and adding cascading for evicts (`CascadeType.EVICT`) during traversals. They suggested evicting `AtomicPart` objects and their to-connection collections after they have been traversed.The Hibernate manual states that "if you are processing a huge number of objects and need to manage memory efficiently, the evict() method may be used to remove the object and its collections from the first-level cache" [76, ch 19.3]. Bauer and King [39] state that "Any persistent object returned by get() or any other kind of query is already associated with the current Session and transaction context. It can be modified, and its state will be synchronised with the database. This mechanism is called automatic dirty checking, which means Hibernate will track and save the changes you make to an object inside a session". Hibernate mentioned that because all the objects being traversed are in Hibernate session *and* the session is checking if objects/entities "have been modified" there is a slight performance overhead. There is also a memory overhead as all the objects are kept in the first level cache in main memory. It was suggested to remove/evict the objects after they have been traversed. While considering and testing the use of evicts it was found that because there is also small modifications to `AtomicPart` objects during a traversal that these modifications need to be *flushed* before evicting the objects. Because *flushing* is needed along with the evicts there was no speed improvement as the extra flushing created an overhead. The recommendation was not activated in the end. It is expected that using evicts along with flushing will assist in running the *medium* and *large* OO7 database configurations. Out of memory exceptions occurred during the initial running of the medium and large OO7 database configurations. By using evicts it is expected to solve the out of memory exceptions that occurred when running these larger configurations. When evicts are implemented in the future it is important to also add evicts for db4o and Versant. Versant and db4o have the same concept as evict.

- Hibernate recommended the use of fetching strategies specifically *subselect* fetching. “A fetching strategy is the strategy Hibernate will use for retrieving associated objects if the application needs to navigate the association”[76, ch 19.1]. Subselect fetching is when “a second SELECT is used to retrieve the associated collections for *all entities* retrieved in a previous query or fetch”. This extra select is only performed when the association is being accessed. Hibernate recommended using subselects on `AtomicPart` objects and their to-connection collections which is the largest collections in the OO7 model. This recommendation was implemented and the `Fetch(FetchMode.SUBSELECT)` annotation was added to `AtomicPart` objects and their to-connection collections.

- The use of cascade deletes for objects involved in the delete operation of OO7 was also recommended by Hibernate. This recommendation was implemented.

- Hibernate recommended using MySQL rather than PostgreSQL because they felt it would perform better. Hibernate configurations using MySQL was added. MySQL version 5.0.51a was used.

- Hibernate suggested using look-up of objects via their id’s. For example `session.load(BaseAssembly.class, baseAssemblyId)` instead of using an HQL query. It was suggested that using look-ups by id would make better use of the cache. Bauer and King [39] state that “the load() method may return a proxy instead of a real persistent instance. A proxy is a placeholder that triggers the loading of the real object when it’s accessed for the first time;” and that “using load() has a further implication: The application may retrieve a valid reference (a proxy) to a persistent instance without hitting the database to retrieve its persistent state” [39]. The Hibernate Entity Manager manual states that “this is especially useful to link a child to its parent without having to load the parent”. So basically the object is retrieved immediately from the cache if present or a proxy is provided for immediate use. Using find/load by object id *would have been useful* during the database creation and linking phase. During this phase `BaseAssembly` objects are retrieved using a query and their id. They are then linked with `CompositePart` objects. Find/load by object id was not implemented for two reasons. First, because the original OO7 uses queries instead of look-ups the study didn't want to deviate from the original OO7. A second consideration was, that *if* find/load by object id was implemented, it would only have been fair to add these look-ups to db4o and Versant as well. So in the end it was deemed a major change and out of scope for the current study. Future work will investigate the use of look-up by id's.

During the review process by Hibernate it was suggested to add checks or assertions into the code to ensure that all the operations performed correctly after adding the recommendations. Because this was on our to-do list this recommendation was implemented. Assertions were added as well as code to check that the correct number of objects were created as well as inserted and deleted by checking the number of objects in the database.

The results using these recommendations will be shown and discussed in section 7.4.

7.2 db4o recommendations

The db4o recommendations were:

- Static db4o configuration classes should not be used anymore because they are deprecated. The static configuration classes also affect the performance of db4o. These static db4o configuration classes were replaced following the recommendation.

- db4o recommended turning off the weak reference cache (setting `config.weakReferences(false)`) during database creation and during insertion of new objects into the database. The db4o reference manual states that "switching weak references off during insert operation releases extra resources and removed the cleanup thread" and that "Weak references need extra resources and the cleanup thread will have a considerable impact on performance since it has to be synchronised with the normal operations within the `ObjectContainer`. Turning off weak references will improve speed" [62]. This setting is very similar to clearing the Hibernate session. The recommendation was implemented.

- db4o recommended changing the default values of the *b-tree* to: `config.bTreeCacheHeight(1000)` and `config.bTreeNodeSize(100)`. The db4o reference manual states that "higher values for the cache height will get you better performance at more RAM consumption".

The default node size in db4o is 100 so only the height is increased from 1 to a 1000. The recommendation was implemented.

- It was recommended that `RandomAccessFileAdapter` and `NonFlushingIoAdapter` should be used. The setup looks as follows: `RandomAccessFileAdapter raf = new RandomAccessFileAdapter();` `config.io(new NonFlushingIoAdapter(raf))`.
  The `NonFlushingIoAdapter` is recommended to "improve commit performance" as "it allows the operating system to keep commit data in cache and do the physical writes in a most performant order (sic)"[62]. While this will improve the inserts there is a risk that the database could be corrupted if a crash occurred. Using the `NonFlushingIoAdapter` is very similar to the Versant setting: `commit_flush off`. The Versant `profile.be` file states that turning `commit_flush off` "specify whether server process buffers are flushed to disk after commits".

- While db4o recommend *not* using eager loading it was included because it providing interesting results.

- db4o recommended that db4o should not be run in client-server mode *(networked mode)* because it is not the *default* mode for db4o, although client-server is the default mode for Versant and Hibernate. Because the focus of this study was on using persistence stores in a enterprise environment only *client-server mode* were considered in the comparisons chapters. db4o embedded mode results were included in chapter 5.

While db4o recommended using the *embedded mode* for db4o in the comparisons, which is the *default mode* for db4o, it was not included in this chapter. It was *deemed unfair* to compare *client-server mode* with *embedded mode.* Future work could include running Hibernate with an embedded database like HSQLDB (HyperSQL DataBase) [20] which could then be compared to db4o running in *embedded mode.* Comparing embedded databases was out of scope.

The comparison between db4o, running in embedded mode and Versant and Hibernate running in client-server mode is included in Appendix A.3 for interest sake.

7.3 Versant recommendations

The Versant recommendations:

- Versant recommended using the default settings provided for the up-coming the Versant version 8.0 release to increase the general performance. This recommendation was not implemented because no new versions were used after a certain cut off date. If one product is upgraded then all the others must be upgraded as well, to be fair.

- The use of gdeletes was recommended because they use less calls to the server, instead batching delete calls. If it were to be used, then batch delete calls would also have to be used for db4o and Hibernate as well, to be fair.

- Versant recommended making sure that indexes are involved during deletes in the benchmark application. It was found that indexes are not directly used during deletes in OO7. Objects that are being deleted do have indexes on some of their fields. It is expected that "index tables" or index structures will be updated during object deletion. The direct use of indexes during deletes were not added as this would be a deviation from the OO7 benchmark.

The next section will show and discuss the results for the new recommended configurations.

7.4 Results when using vendor recommendations

In this section the results of the different implementations are compared and investigated. The *original* results and the new results obtained using the vendor *recommendations* are included on the graphs.

Again the results are grouped into creation, traversal, query and modification sections.The results will not be discussed in the same depth as in the previous sections and chapters as this extra work was not the focus of the study. Only some of the results that stand out will be discussed.

7.4.1 Creation

By using the recommendations provided by db4o and Hibernate the creation time have definitely improved.

The db4o's *eager* loading configuration has a 38% improvement and db4o's *lazy* loading configuration has a 79% improvement. Hibernate has a ±36% improvement. The newly added MySQL configuration is ±2 seconds faster than PostgreSQL.

Overall Versant is still the fastest followed by Hibernate.

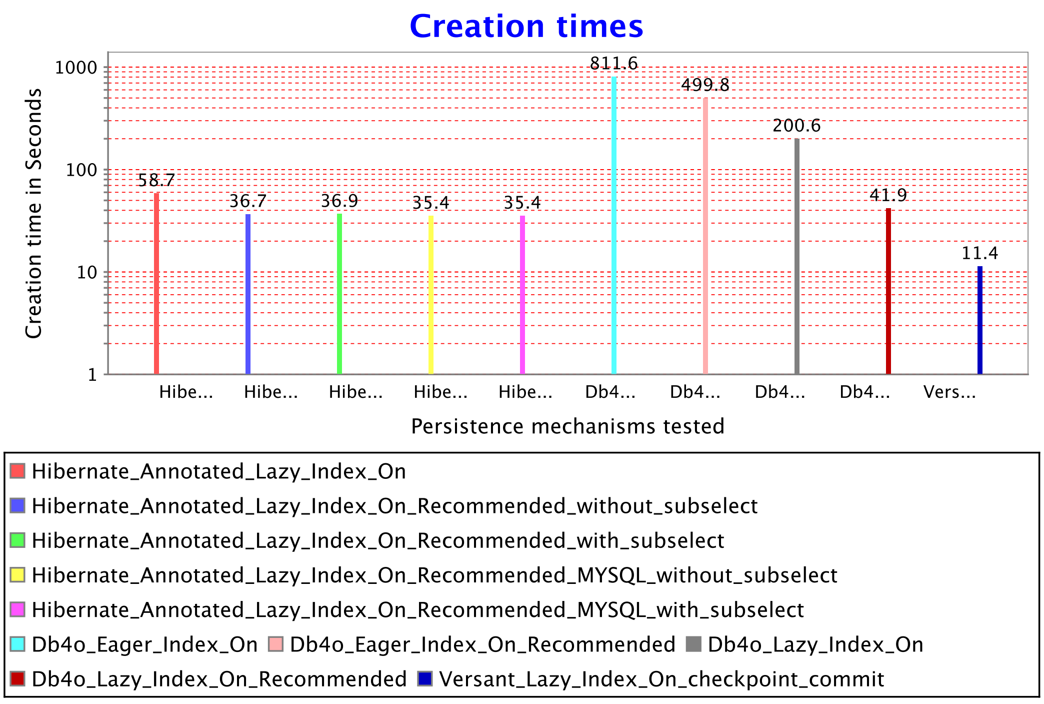


Figure 7.1: Creation times

7.4.2 Traversals

Inspection of the graphs shows that the recommendations have improved traversal times. For example the cold traversal time for *Trav2c* using the new db4o lazy loading configuration has gone down from 45 seconds to 31 seconds. This is a 31% improvement.

Hibernate's time for *Trav1* has gone down from 300 seconds to 87 seconds in the cold run for the PostgreSQL configuration using no subselects. This is a 71% improvement. The MySQL configuration using *no subselects* performs the best of the Hibernate configurations for traversals that involve a large number of objects and traversals with modifications. These traversals are *Trav1, Trav2a, Trav2b, Trav2c, Trav3a, Trav3b* and *Trav3c*. It is surprising to find that adding *subselects*, as recommended by Hibernate, actually slows down the MySQL configurations. For *Trav1* in the cold run the MySQL configuration with a subselect slows down from 15 seconds to 39 seconds. This is a 160% decrease in speed. An investigation into the reasons why *subselects* on MySQL is slow was regarded as beyond the scope of this study.

For the Hibernate configuration using PostgreSQL and subselects the traversal time in the cold run for *Trav1* has gone down from 87 seconds to 17 seconds. This is a 80% improvement. The 80% improvement is also evident for the traversals *Trav2a, Trav2b, Trav2c, Trav3a, Trav3b* and *Trav3c.*

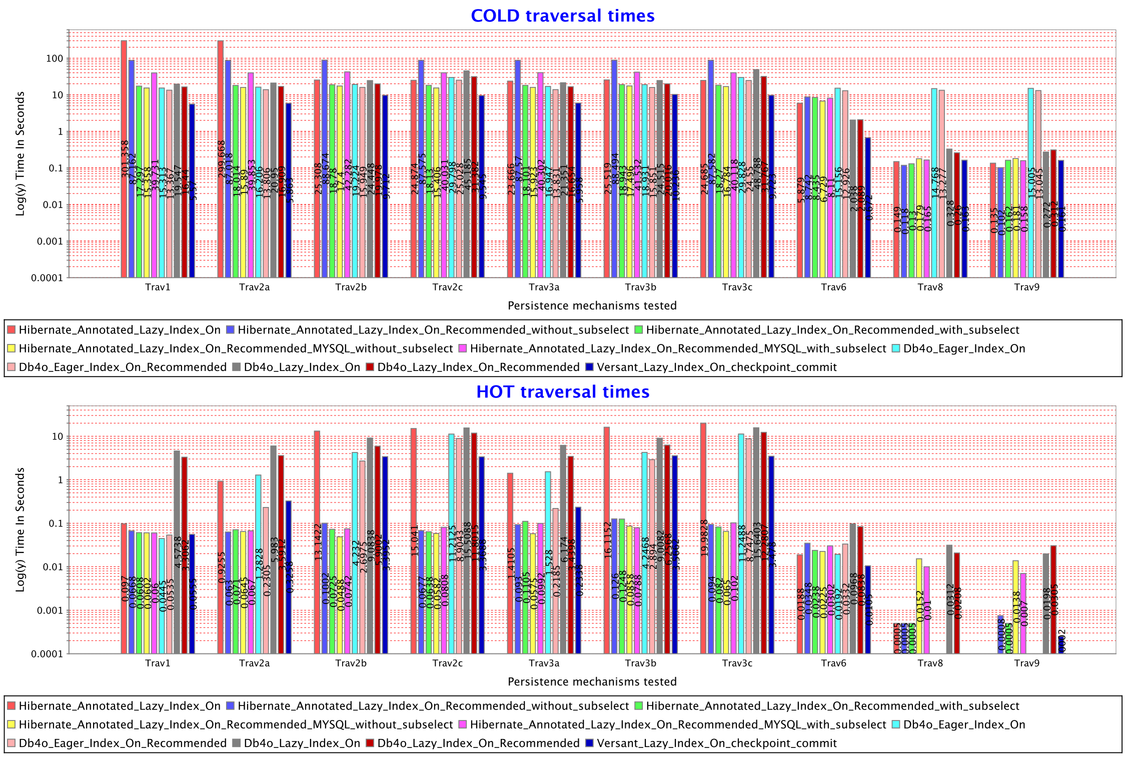


Figure 7.2: Comparison results of traversals

Another surprising result is that the original Hibernate configuration on PostgreSQL (the red bar) is faster in *cold* traversals *Trav2b, Trav2c, Trav3a, Trav3b* and *Trav3c* than the newly recommended Hibernate configuration on PostgreSQL with *no subselects* (the light blue bar). On average the original Hibernate configuration on PostgreSQL take about 25 seconds to complete. The newly recommended Hibernate configuration on PostgreSQL with no subselects takes about 87 seconds to complete. This means that the newly recommended Hibernate configuration is 248% slower. Hibernate suggested a possible explanation for this. They suggest that because data is duplicated in the old configuration and because foreign keys (FK) are duplicated as well, traversals can run faster because there is no need to "hit the inverse side" of an association. Also all the data "is accessible from both sides" of the association. They also state that by using the *table per class (joined)* strategy, the joins do not cause a large overhead as expected because there is only one level of inheritance in the OO7 model. If there were more inheritance levels the performance is expected to decrease (get slower) when using the *table per class (joined)* strategy for assemblies.

In cold runs the original Hibernate configuration on PostgreSQL (the red bar) is slower than the other Hibernate configurations in all cases except for *Trav6, Trav8* and *Trav9* where a small number of objects are being traversed.

Versant is still the fastest for all the cold traversals. The only exceptions are again *Trav8* and *Trav9* where the Hibernate configurations are fastest.

Looking at hot traversals the new Hibernate annotation configurations using the recommendations are now faster than the db4o and Versant configurations in most cases (6 out of 10). This is a reversal from the results shown in the previous chapter.

Taking a look at the observations made in the previous chapter the following is noted:

- Cold runs: It is still true that for *Trav1*, both db4o's lazy configuration and Hibernate's lazy configurations are slower than the the eager db4o configuration and Versant. Also note that *Trav1 and Trav2b* traverse the *same number* of objects but the objects are *modified* in *Trav2b*. When objects are changed in *Trav2b* it is not true anymore that Hibernate times improve compared to the Hibernate times for *Trav1*. The Hibernate times are now slower. Only the original Hibernate configuration on PostgreSQL improve (the red bar) when modifications are made. For *Trav1* the run times were around 300 seconds and are now around 25 seconds. The reason for this could be that the changes are not flushed immediately or it could related to the way in which the model is mapped. db4o and Hibernate's times are very slow for *Trav1* *(*traverse 43 700 objects*)* compared to Versant. db4o's times are between 13 and 19 seconds and Hibernate is between 15 and 87 seconds compared to Versant's 5 and 9 seconds. *It is still true, that when many objects are traversed and modified in the cold Trav2b run, that Versant scales better.*

- Looking at *Trav1* and *Trav2b* times in the hot runs, it is observed that the times now increase for all configurations when running and modifying objects in *Trav2b* compared to their times when merely traversing the objects in *Trav1.* This is still the case when running the new recommended configurations. There is a reversal now in that Hibernate is now faster than db4o and Versant.

- For *Trav2a* and *Trav3a* in the hot runs, which involve changes to a small number of objects (2185), Hibernate and the db4o eager loading configuration are now slightly faster than Versant. The Hibernate configurations on MySQL performs *Trav2a* in 0.06 seconds while Versant takes 0.3 seconds.

- In the previous chapter it was mentioned that in traversal *Trav3c* the Hibernate cache performs better than db4o on multiple immediate repeats, while in *Trav3b* db4o performs slightly better. Now the new Hibernate configurations are faster in both Traversal 3 and Traversal 2 in the *hot runs*.

- In larger traversals like *Trav2c* and *Trav3c*, Versant is still the fastest for the cold runs. In the *hot runs*, the new improved Hibernate annotation configurations with named queries, are now the fastest, overtaking Versant. If once-off traversals (cold runs) are needed, that involve large number of objects, Versant performs better. The improved Hibernate configurations are recommended when repeated access is needed and caching can be used.

- Hot and cold runs: Carey et al. [45] mention that times increase between *Trav2a* and *Trav2*b because the number of objects being traversed and updated increases. The same is true for *Trav*3b and *Trav*3c. Looking at Figure 7.2 it is still true that the db4o traversals times increase moving from *Trav2*b to *Trav2*c. The db4o eager loading configuration time increases from 2 seconds for *Trav2*b to 8 seconds for *Trav2*c in the hot run. The db4o lazy loading configuration time increases from 5 seconds for *Trav2*b to 11 seconds for *Trav2*c in the hot run. Figure 7.2 further shows that Hibernate's and Versant's times actually remain consistent between *Trav2*b to *Trav2*c.

- It is still true that in all the hot runs db4o's lazy loading or manual lazy activation seems to create an overhead compared to the eager loading configuration. For example in *Trav2a* the db4o eager loading configuration runs on average ± 0.2 seconds while the db4o lazy configuration takes ± 3.5 seconds.

- For *Trav6* in the cold run Versant is still the fastest at 0.6 seconds. db4o lazy loading configurations are second at ± 2 seconds followed by Hibernate which is between 8 and 5 seconds. *Trav6* traverses the same number of objects as in *Trav2a* and *Trav3a* but with no updates to any fields.

- For *the hot Trav6* run, Versant is still slightly faster than Hibernate followed by db4o.

In the next section the impact of the recommendations on the queries will be investigated.

7.4.3 Queries

Figure 7.3 shows the new query results. Looking at queries it is clear that the Hibernate recommendations have slightly improved the performance of the query runs. The biggest improvement for Hibernate is Query 8. Query 8 has gone down from around 3000 seconds to about 1300 seconds. This is a $\pm$ 56% improvement.

For db4o the db4o eager loading configuration has improved the *cold* Query 8 speed from 5800 seconds down to about 3900 seconds. This is a $\pm$ 33% improvement.

Versant is still the fastest for all the cold and hot query runs. Hibernate's times are now closer to Versant's times.

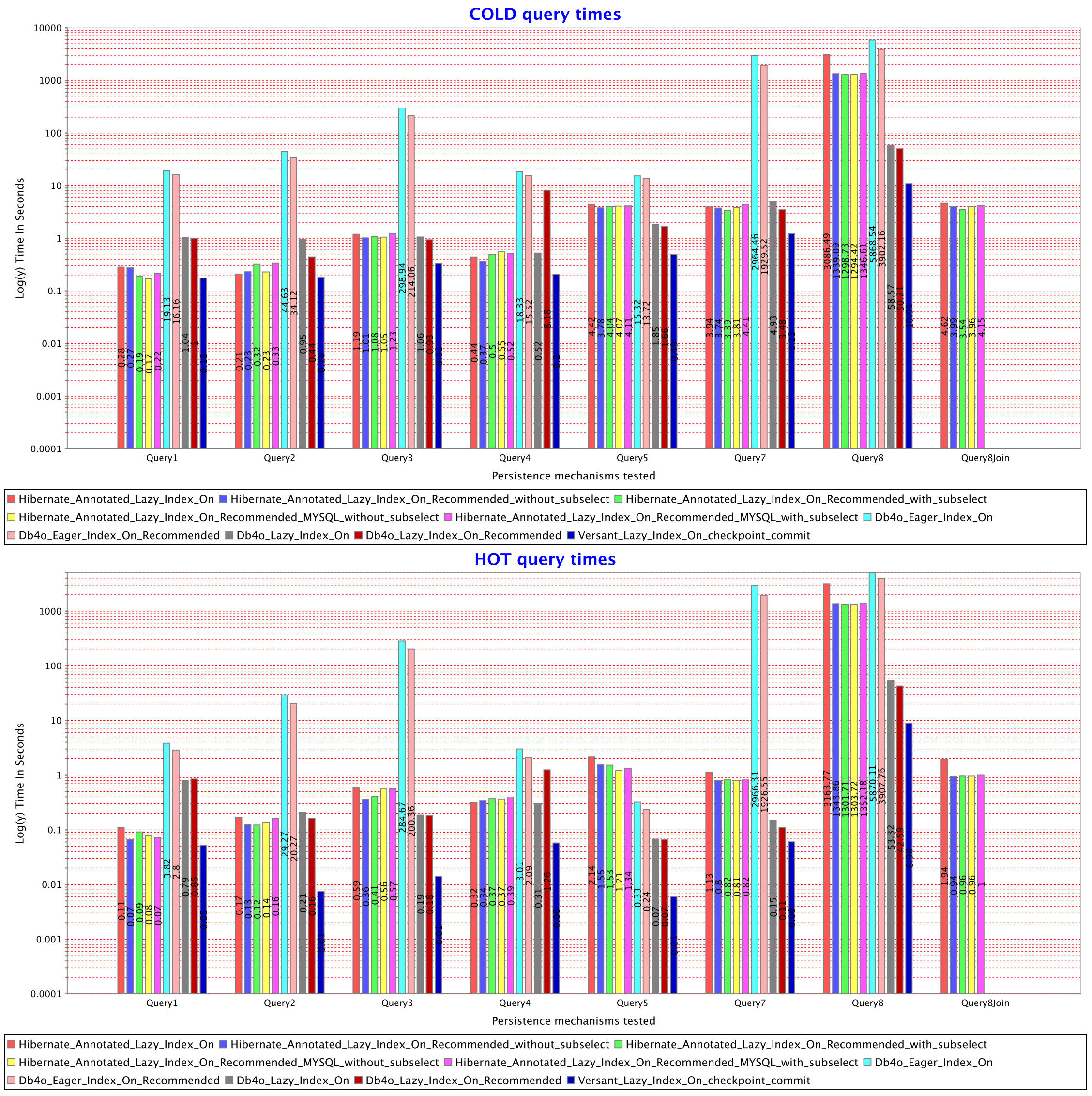


Figure 7.3: Comparison results of queries

Taking a look at the observations made in the previous chapter it is noted that:

- Query 1: Hibernate and Versant still perform this query, which is on an index field, faster than db4o in the hot and cold runs. db4o using eager loading now performs Query 1 in ±16 seconds and db4o using lazy loading takes still takes around 1 second. Versant and Hibernate perform Query 1 in less than 0.3 seconds.

- As explained in the previous chapter: Query 2 and Query 3 are range queries. Because Query 3 finds more objects (finds 10%) it will be slower than Query 2 (only finds 1%). Figure 7.3 shows that Query 3 still remains slower than Query 2. db4o's lazy configuration times are still slightly faster than Hibernate in Query 3 in the *hot* run, with this increase of objects. Hibernate still increases by almost 1 second between Query 2 and Query 3 in the cold run while db4o lazy recommended configuration increases now by ± 0.5 seconds.

- For Query 5 db4o's eager configuration times improve from 13 seconds to 0.2 seconds in the hot run.

- For Query 7 db4o's lazy configuration still improves in the hot run to be ahead of the Hibernate configurations. db4o's lazy configuration performs Query 7 in the hot run in 0.1 seconds while Hibernate performs it in 0.8 seconds.

Overall, Hibernate is faster than db4o in the following queries: Query 1, Query 2, Query 4, Query 7 and in Query 8 using the the *join* query*(*Query8Join). Thus Hibernate is faster than db4o in 5 out of the 7 queries in the cold runs.

Figure 7.4 shows the speed improvement due to the cache for queries and traversals. For all the recommended runs the cache usage has improved slightly.

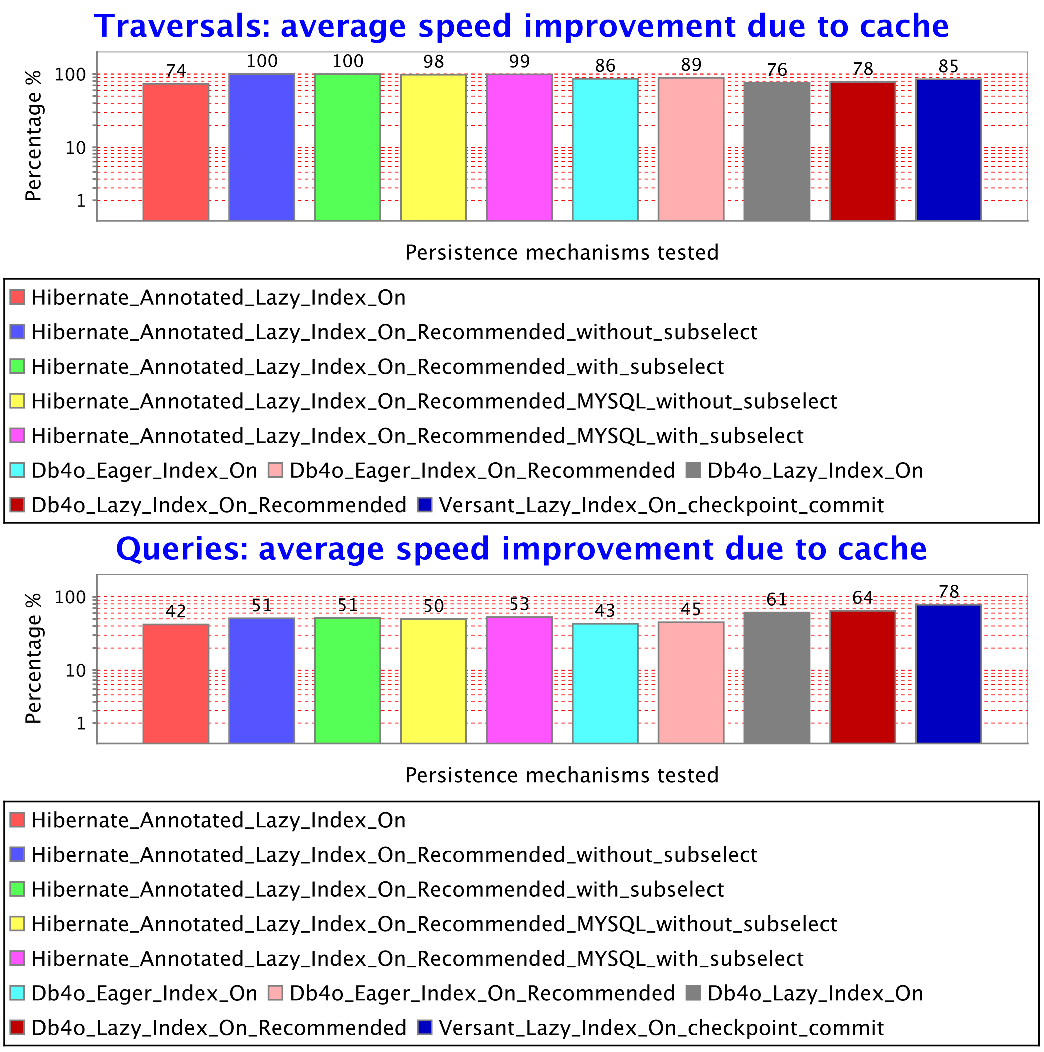


Figure 7.4: Comparing traversal and query average speed improvement due to cache when using vendor recommendations

7.4.4 Modifications: Insert and delete

Figure 7.5 and Figure 7.6 shows the new insert and delete results. The recommendations have improved these times.

Looking at inserts the following is observed:

- The db4o eager loading configuration have improved from 4.7 seconds to about 3.2 seconds. This is a 32% improvement. The db4o lazy configuration has also improved from 0.9 seconds down to 0.27 seconds. This is a 70% improvement.

- Hibernate on PostgreSQL has improved from 1.1 seconds to 0.9 seconds in the hot run. This is a 18% improvement.

Versant remains the fastest for both cold and hot inserts.

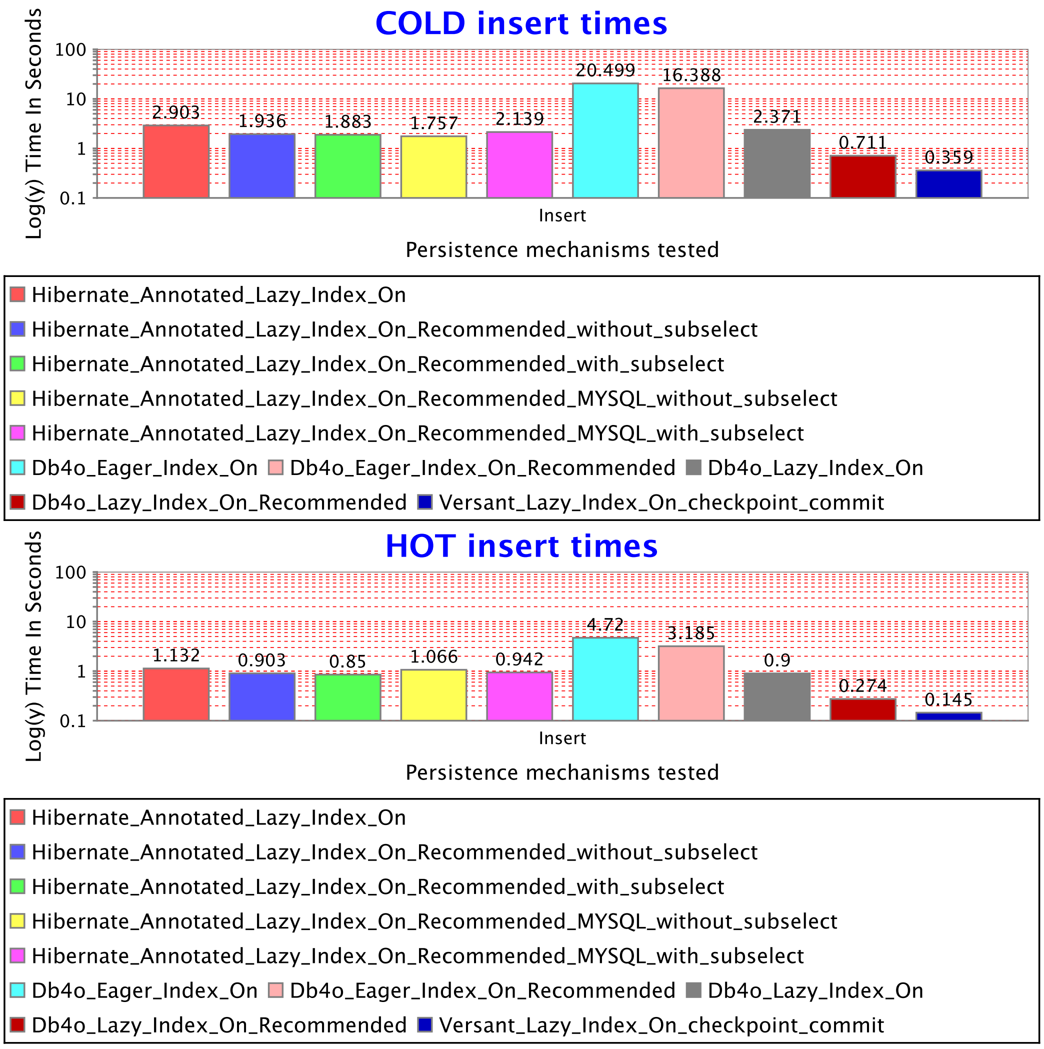


Figure 7.5: Comparison result of inserts

The following is observed for the Hibernate configurations during deletions:

- The Hibernate configurations running on *MySQL* perform *deletes* faster than the Hibernate configurations running on *PostgreSQL*.

- The Hibernate configuration on PostgreSQL originally took 39 seconds in the cold run and now takes around 4 seconds. There is a 90% speed improvement for deletes using the Hibernate recommendations running on *PostgreSQL*.

db4o remains the fastest for deletes.

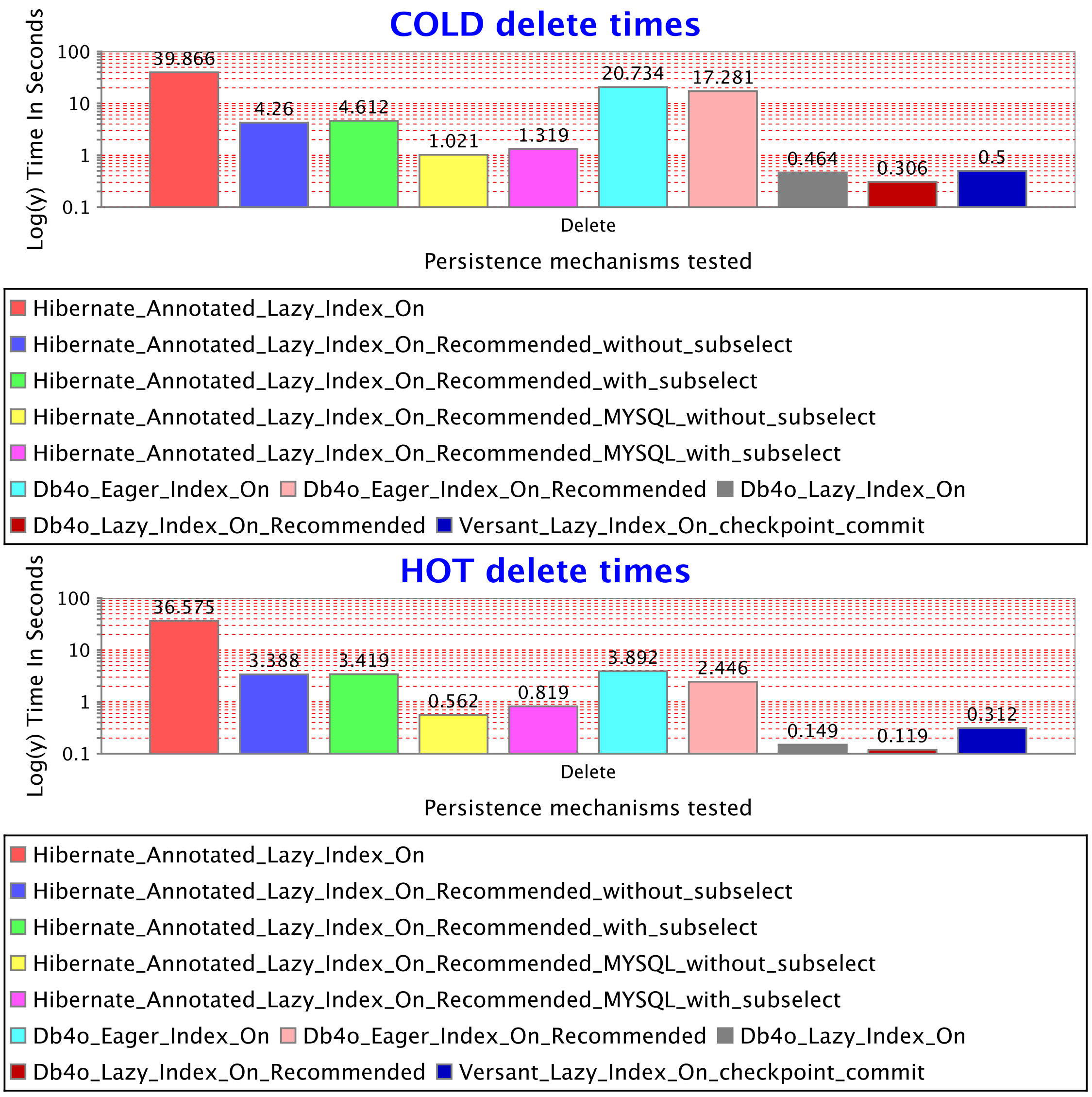
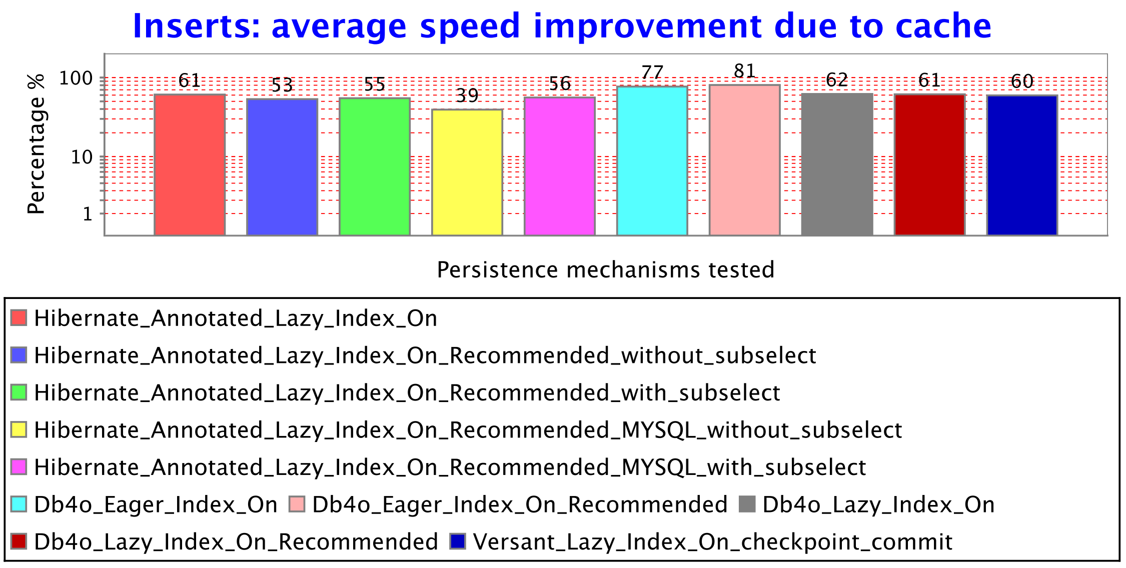


Figure 7.6: Comparison result of deletes

Figure 7.7 below shows the speed improvement due to the cache.

Using the new recommendations it is observed that the cache improvement has increased for deletes and decreased slightly for inserts.



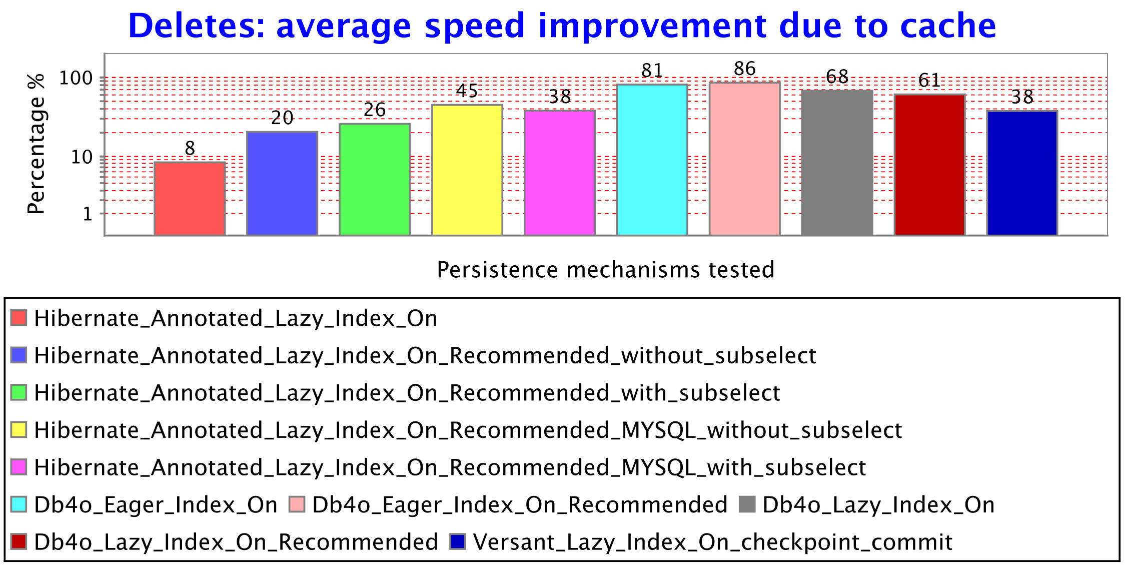


Figure 7.7: Comparing insert and delete average speed improvement due to cache for all

7.5 Scoring the benchmark results

This section will again use the *number of first places* on the benchmark and the *weighted ranking* of places to rank the different configurations. In this section only client-server configurations are used. It is deemed unfair to compare embedded mode with client-server. Appendix A includes, for interest sake, results where db4o embedded mode has been compared to client-server configurations. This comparison is not recommended.

7.5.1 Number of first places ranking

The number of first places using the new recommended configurations results are provided in Table 7.1. As stated earlier in this chapter the Hibernate XML mapping configuration was excluded from the benchmark results in this chapter. The fastest Hibernate configuration was selected. The previous rankings from chapter 6 are provided in brackets. The Hibernate configuration, running on MySQL, is a new entry so it does not have a previous ranking. From the table below it is clear that the Hibernate MySQL configuration perform better than db4o. Versant is still best overall.

Rank	Persistence Mechanism/Object database/ ORM Tool	Score
1 (1)	Versant	23
2 (-)	Hibernate using annotations with lazy loading, indexes on and named queries. MySQL with no subselect	8
3 (3)	db4o with eager loading and indexes on *Using recommendations*	3
4 (4)	db4o with lazy loading and indexes on *Using recommendations*	2
5 (2)	Hibernate using annotations with lazy loading, indexes on and named queries. PostgreSQL and subselect	2

Table 7.1: Ranking according to the number of first places

7.5.2 Weighted ranking

Weighted ranking of places results are shown in Table 7.2. The Hibernate configuration running on PostgreSQL using a subselect is now in 3rd place. The configuration running on MySQL and using no subselect is in 2nd place. db4o's eager configuration was third in the *first place rankings* but is now in 5th place.

Rank	Persistence Mechanism/Object database/ ORM Tool	Score
1 (1)	Versant	65
2 (-)	Hibernate using annotations with lazy loading, indexes on and named queries. MySQL with no subselect	98
3 (2)	Hibernate using annotations with lazy loading, indexes on and named queries. PostgreSQL and subselect	117
4 (3)	db4o with lazy loading and indexes on *Using recommendations*	143
5 (4)	db4o with eager loading and indexes on *Using recommendations*	147

Table 7.2: Ranking according to weighted ranking of places

As mentioned in the previous chapter the foregoing assessment is rather course-grained. As an alternative, the weighted ranking results may be computed at a finer level of granularity. Consequently, weighted ranking results have been computed separately for the cold and hot runs. Additionally, ranking results have also been broken down separately for traversals and queries. These results are shown in the following subsections.

7.5.2.1 Weighted ranking for cold runs

If the application under consideration operates mostly on data that is *not repeated* and mostly *on disk* then, the cold run ranking is very important. The weighted ranking results for just cold runs are shown in Table 7.3 below. The db4o lazy loading configuration has now moved down to 4th place. The Hibernate configuration running on MySQL and using no subselect is now in 2nd place. The db4o configurations are now last. Versant is still the best overall.

Rank	Persistence Mechanism/Object database/ ORM Tool	Score
1 (1)	Versant	24
2 (-)	Hibernate using annotations with lazy loading, indexes on and named queries. MySQL with no subselect	53
3 (2)	Hibernate using annotations with lazy loading, indexes on and named queries. PostgreSQL and subselect	62
4 (3)	db4o with lazy loading and indexes on *Using recommendations*	68
5 (5)	db4o with eager loading and indexes on *Using recommendations*	78

Table 7.3: Ranking according to weighted ranking of places for just the cold run

7.5.2.2 Weighted ranking for hot runs

If the application under consideration operates mostly on data that is *repeatedly accessed* and mostly *in-memory* then the hot run ranking is very important. The weighted ranking results for just hot runs are shown in table 7.4 below. Overall, the Hibernate configurations *still* perform better than the db4o configurations in this situation. The Hibernate rankings are now closer to Versant.

Rank	Persistence Mechanism/Object database/ ORM Tool	Score
1 (1)	Versant	41
2 (-)	Hibernate using annotations with lazy loading, indexes on and named queries. MySQL with no subselect	45
3 (2)	Hibernate using annotations with lazy loading, indexes on and named queries. PostgreSQL and subselect	55
4 (3)	db4o with eager loading and indexes on *Using recommendations*	69
5 (4)	db4o with lazy loading and indexes on *Using recommendations*	75

Table 7.4: Ranking according to weighted ranking of places for just the hot run

7.5.2.3 Weighted ranking for cold query runs

If the application under consideration operates mostly on data that is *not repeated*, mostly *on disk* and involves queries, then the weighted ranking results based only on cold query runs in Table 7.5 below are relevant. It is important to note that when calculating these scores that the new Hibernate *Query8Join*, using a join query, was used used in the ranking calculations.

The new Hibernate configuration on MySQL with no subselects have taken over 2nd place. db4o's lazy loading configuration still performs better than the db4o eager loading configuration for queries in cold runs.

Rank	Persistence Mechanism/Object database/ ORM Tool	Score
1 (1)	Versant	10
2 (-)	Hibernate using annotations with lazy loading, indexes on and named queries. MySQL with no subselect	18
3 (2)	Hibernate using annotations with lazy loading, indexes on and named queries. PostgreSQL and subselect	19
4 (4)	db4o with lazy loading and indexes on *Using recommendations*	23
5 (5)	db4o with eager loading and indexes on *Using recommendations*	35

Table 7.5: Ranking according to weighted ranking of places for just the cold query runs

7.5.2.4 Weighted ranking for hot query runs

If the application under consideration operates mostly on data that is *repeatedly accessed*, is mostly *in-memory* and involves queries then the weighted ranking results limited to hot query runs in Table 7.6 below are of interest. The Hibernate configuration on PostgreSQL with subselects is tied with the db4o lazy loading configuration in 2nd place.

Rank	Persistence Mechanism/Object database/ ORM Tool	Score
1 (1)	Versant	9
2 (-)	Hibernate using annotations with lazy loading, indexes on and named queries. MySQL with no subselect	19
3 (2)	db4o with lazy loading and indexes on *Using recommendations*	22
4 (3)	Hibernate using annotations with lazy loading, indexes on and named queries. PostgreSQL and subselect	22
5 (5)	db4o with eager loading and indexes on *Using recommendations*	33

Table 7.6: Ranking according to weighted ranking of places for just the hot query runs

7.5.2.5 Weighted ranking for cold traversal runs

If the application under consideration operates mostly on data that is *not repeated*, mostly *on disk* and involves traversals then the weighted ranking results, for just cold traversal runs in Table 7.7 below are relevant. The db4o eager configuration is still in 3rd place but ahead of the Hibernate configuration running on PostgreSQL using a subselect.

Rank	Persistence Mechanism/Object database/ ORM Tool	Score
1 (1)	Versant	11
2 (-)	Hibernate using annotations with lazy loading, indexes on and named queries. MySQL with no subselect	28
3 (3)	db4o with eager loading and indexes on *Using recommendations*	33
4 (2)	Hibernate using annotations with lazy loading, indexes on and named queries. PostgreSQL and subselect	36
5 (4)	db4o with lazy loading and indexes on *Using recommendations*	42

Table 7.7: Ranking according to weighted ranking of places for just the cold traversal runs

7.5.2.6 Weighted ranking for hot traversal runs

Finally, if the application under consideration operates mostly on data that is *repeated*ly *accessed*, is mostly *in-memory* and involves traversals, then the weighted ranking results limited to hot traversal runs in Table 7.8 below should be considered. Versant has fallen down to 4th place. The Hibernate configurations are now in 1st and 2nd place followed by the db4o eager loading configuration.

Rank	Persistence Mechanism/Object database/ ORM Tool	Score
1 (3)	Hibernate using annotations with lazy loading, indexes on and named queries. PostgreSQL and subselect	19
2 (-)	Hibernate using annotations with lazy loading, indexes on and named queries. MySQL with no subselect	25
3 (2)	db4o with eager loading and indexes on *Using recommendations*	27
4 (1)	Versant	29
5 (5)	db4o with lazy loading and indexes on *Using recommendations*	50

Table 7.8: Ranking according to weighted ranking of places for just the hot traversal runs

7.6 Summary

The vendors had a limited time to review the benchmark results and while they have provided a list of recommendations there are still other optimisation techniques that have not been investigated. For example not *all* the Hibernate *fetch strategies* have been investigated. Such an investigation is left as future work. Also, given more time and using more of the vendor recommendations it should be possible to improve the results even more.

Overall Versant stayed in first place. With the Hibernate recommendations the Hibernate performance has improved a lot. While the db4o results have improved it is clear that the new Hibernate recommendations are still faster in most cases.

It is important to note that this chapter, which includes more fine tuning using the vendor recommended optimisations, was added as a matter of curiosity and was not the main focus of the study.

The next chapter will provide a final summary and conclusion of the study.

Chapter 8

Summary, conclusions and future work

This chapter provides a summary of the study and the conclusions reached during the study. A list of future work is also provided.

8.1 Summary and conclusions

The OO7 benchmark is classified as a *research, micro benchmark* with a *synthetic* load that tries to simulate real world loads and operations. As stated in earlier chapters it is important that benchmarks should be *repeatable*, *relevant*, use *realistic* metrics, be *comparable* and are widely used [91]. OO7 was selected for this study as it complied to the above criteria.

It is important to note that this study has taken Versant, db4o and Hibernate with PostgreSQL and used them in their *default modes* (out of the box). Certain performance techniques and settings were then changed and documented. Using Versant, db4o and Hibernate with PostgreSQL *out of the box* meant that *internally* they could have settings turned on that might improve their performance. The study tried to be as fair as possible and transparent as possible as to what settings were changed. It is also important to understand what the defaults are. Also if any of these settings are changed it must be reported.

The study for example did not investigate the following settings that might influence performance:

- Server side cache settings. For example in Versant the server page cache size could be set [100, page 270] and could influence performance.
- Flushing strategies. Hibernate only synchronises changes to objects in

the Session to the database when commit is called or an explicit flush is called [39, p 160]. Hibernate provides settings that could change when this flush occurs. db4o has similar settings. In db4o "switching off flushFileBuffers can improve commit performance as the commit information will be cached by the operating system" [62].

- Locking modes. Some of the databases and ORM Tools are able to switch between pessimistic and optimistic transaction locking modes.
- Transaction logging can be turned on and off. If turned off, the speed could increase but if a crash occurs the transaction changes are lost.
- Different query modes and query fetching strategies. Hibernate provides different fetching strategies that can improve the performance of Hibernate when objects are retrieved and queried. Some of these fetching strategies are join fetching, lazy collection fetching, etc. [76]. db4o provides different query modes that could influence performance. These modes are immediate, lazy and snapshot query modes [62].

The study had a few research objectives. These were:

- Investigate the *performance* of object databases against that of ORM tools.
- Find how different optimisation techniques influence the performance.
- Understand *when to use* an object database or an ORM tool by investigating which of them perform better for different types of operations. This was discussed in sections 6.2 and 7.5.
- Part of the study was also to investigate the OO7 benchmark in greater detail to get a clearer understanding of the OO7 benchmark code and inside workings thereof. Chapter 3 discussed for example the number of objects that are created, traversed and queried by the OO7 benchmark in *great* detail. The *process* to create and run the OO7 model and benchmark operations was also discussed in great detail in Chapter 3.

The performance of newer open source object databases was investigated and compared to that of older more established proprietary object databases. *It was found that Versant which is an established proprietary object database performed better than db4o*, which is an open source object database, in most cases.

As stated *optimisation techniques* were also investigated. The investigation wanted to find how much certain techniques improve performance and in what circumstances could or should these techniques be used.

The techniques that were investigated are:

- Indexes
  - It was found that the use of an index does help for queries. This was expected. An example where this is clearly seen is Query 1, which uses an index during look-up. The results for using or not using an index are show in Chapter 5. Hibernate's indexes seem to function better than db4o's indexes during queries. This can be seen in Figure 5.11 where for db4o, turning an index on and off for Query 1 is very close in the hot run where the cache assists to improve the speed. In the cold run having an index actually makes db4o slightly slower.
- Lazy and eager loading
  - The differences between lazy loading and eager loading have been shown in Chapter 5 and Chapter 6. Eager loading improves traversal times if the tree being traversed can be *cached* between iterations. Although the first run can be slow, by caching the whole tree, calls to the server in follow-up runs are reduced. Lazy loading improves query times and unnecessary objects are not loaded into memory when not needed.
- Caching
  - It was found that caching helps to improve most of the operations.
  - By using eager loading and a cache, *the speed* of repeated traversals of the same objects is increased.
  - Hibernate with its first level cache in some cases perform better than db4o during traversals with repeated access to the same objects. This is shown in section 7.4.2.

In the end the performance difference between using one transaction versus many transactions to run the OO7 benchmark operations was not included in the study because of time constraints. The use of different types of queries, query modes and query fetch strategies was also not investigated. These omissions will be included in future work. Future work could include creating test configurations where these settings and their impact are investigated.

Chapter 7 included the performance settings recommended by the vendors. Some of the db4o settings were:

- B-tree settings can be changed when used for queries. db4o "uses special B-tree indexes for increased query performance and reduced memory consumption" [62] and provides settings for changing the B-tree's cache height and the B-tree nodes sizes. These could influence the performance of the queries.
- Caching on the client side could be turned off. In db4o the weak reference cache can be turned on and off. "Weak references need extra resources and the cleanup thread will have a considerable impact on performance since it has to be synchronised with the normal operations within the ObjectContainer. Turning off weak references will improve speed" [62].

Versant has a tuning wizard (`vdbadmin`) whereby the following settings could for example be changed to influence the performance [15]:

- The amount of memory available for the database at start-up.
- The database size.
- The processing types.
  - Commit intensive: Lots of smaller concurrent transactions, optimises transaction throughput.
  - Batch intensive: Smaller number of transactions making lots of changes, optimises for data throughput.
- Application types or profiles.
  - Development only.
  - Read/Query intensive.
  - Update intensive.
  - Create intensive .

None of these settings were changed and the default out of the box settings were used for Versant.

In Carey et al. [47] they reflect on the OO7 benchmarking effort and discuss some of the issues they have experienced. Looking at these, the current study has experienced similar issues:

- They mention that they were critiqued for not *using all the performance enhancing techniques* offered by the databases they were benchmarking. They state in their defence that they did not want to make the benchmark effort a tuning contest "for wizards" or vendors but rather a comparison effort of the engines and how a normal "intelligent, but not exceptional, application programmer" would use them. The current study shared their objective of using the products in their *"out of the box"* state as a normal programmer would. Chapter 7 included vendor recommendations. These recommendations were provided by experts and it is used to compare the vendor recommendation results with the base, *out of the box* results created by a normal programmer. The vendor recommendations have improved the overall performance. With the use of the base results in Chapter 6 it is now possible to measure the improvements. If there was no base to compare with, and if all the performance enhancing techniques were used the first time, there would be no way to see which techniques influenced the performance.

- They ask the question of who should audit the auditors/benchmarkers. They suggest than benchmark results should be reviewed by an independent body, for example a standards organisation. There are organisations like TPC (Transaction Processing Performance Counsel) that are independent and manage the TPC database benchmarks. Today there is still no organisation handling the OO7 benchmark or benchmarks created for measuring object database and ORM tools performance. The approach currently taken for this study is to provide the code of the benchmark as open source and to make it possible for outsiders and experts to review what has been included in the benchmark.

- "Freedom of information". Many of the licenses at that time (1990's) *prohibited* benchmark results from being published without permission from the vendors. This issue is still true today and permission to release Versant results was needed. Hibernate, PostgreSQL and db4o have no such provision in their open source licenses.

When creating and using benchmarks it is important to clearly state what settings and environment is being used. In this study it was found that:

- Running in *client-server* mode or *embedded* mode in db4o has different performance results. In Chapter 5 it was shown that some queries are faster when db4o is run in embedded mode and also that some traversals are faster when db4o is run in client-server mode.

- That it is important to state when a *cache* is being used and cleared. In Chapter 5 it was shown that clearing the Versant cache with every transaction commit influenced the hot traversal times. It is also important for example, to state if a first or second level cache is being used in Hibernate as this could influence traversal times.

- Having a cache does not always improve random inserts, deletes and queries. In Chapters 5, 6 and 7 it was shown that the cache assisted traversals more than inserts, deletes and queries. Because of random inserts, deletes and queries not all objects accessed will be in the cache. Also if query caches are used, it must be stated clearly, otherwise it could create false results.

- It is important to note that it is quite difficult to generalise. One mechanism is faster with updates to smaller amount of objects, others with larger amount of objects; some perform better with changes to index fields others with changes to non indexed fields; some perform better if traversals are repeated with the same objects. Others perform better in first time traversals which might be what the application under development needs. The application needs to be profiled to see if there is *repeated access* to the same objects in a client session.

- In most cases the cache helps to improve access times. But if an application does not access the same objects repeatedly or accesses scattered objects then the cache will not help as much. In these cases it is very important to look at the OO7 cold traversal times which access the *disk resident* objects. For cold traversals, or differently stated, first time access to objects, in most cases Versant is the fastest of all the mechanisms tested by OO7.

- By not stating the benchmark and persistence mechanisms settings clearly it is very easy to cheat and create false statements. It is easy to cheat if certain settings are not brought to light. For example with Versant it is important to state what type of commit is being used: a normal commit or a checkpoint commit.

- It is important to make sure that the performance techniques that are being used actually improve the speed of the application. It has been shown that to add subselects to a MySQL database does not automatically improve the speed.

OO7 benchmark results could be used to provide a recommendation. Although the benchmark results are used to provide recommendations, it is important to understand that a benchmark only provides a guide and that *real life applications* must be tested and benchmarked with the different options.

This study started with initial assumptions that object databases will be faster and that ORM tools will not come close. This study has shown that Hibernate, an ORM tool, for example, performs better in most cases than db4o when objects are traversed (Chapter 6 and Chapter 7). This is especially true in hot runs where the traversals are *repeated* on the *same* objects in memory. It was found that having a cache improves performance times. Hibernate's first level cache performs well for object traversals in hot runs where the traversal are *repeated* on the *same* objects in memory. If hot query runs are performed db4o performs slightly better than Hibernate. This is shown in section 7.5.2.4. Chapter 7 has shown how the vendor recommendations improved the performance of Hibernate, Versant and db4o. It was also found that Versant was faster than Hibernate in most cases when performing certain types of queries. But when large queries were rewritten, like Query 8 to make it a join like query, Hibernate performed better.

In conclusion looking at the results it is important to investigate what type of operations are important for the application under development and to benchmark to find out which of the ORM tools and object databases are the best fit with the application. It is also important to be wary of statements that state that one product is better than the other. These statements need to be investigated and tested for the specific application.

8.2 Future work

Currently there is an ongoing study in which *some* of this future work will be included. The following future work has been identified and will be included in the ongoing study:

- Create multi-user OO7 benchmark as suggested by [47]. This means creating more simultaneous clients running operations on one database. Includes tests for concurrency, scaling and performance.

- Create one Java class model that can be included by all implementations. Currently there is an implementation for each of the the products being benchmarked with the common model being duplicated. The reason for this is discussed in Chapter 3.

- Currently the utility classes are implemented using static methods. This means that there is only one handle to the underlying persistent store. On reflection this was *not a good design choice* and will be changed in future.

- Run medium and large OO7 benchmark configurations. The use of the large and medium OO7 benchmark configurations were investigated. It was found that they needed more processing power and memory to run than were available for the study. The medium database has about 400 000 objects and the large database configuration 4 million objects. It took days to create and run the operations. Many of the database configurations failed to complete because of memory issues. In the end these configurations were not used because the necessary hardware resources were not available for this study. It is estimated that the database sizes would be between 250 MB and 2 GIG. These are not large databases by today's standards. The transaction strategy used in creating and running these larger benchmark operations is also very important. If one long transaction is used that keeps objects in memory, out of memory errors might occur. Also some databases allow the database file to be pre-grown which could help to improve the database creation time. These and other strategies will be investigated in future work as well. Future work will included revisiting and re-running these larger OO7 benchmark configurations.

This study leaves the following matters for future researchers to consider:

- Upgrade the OO7 benchmark configurations to create larger tera or peta byte data sets to investigate large database performance. The large and medium database sizes are small by today's standards and need to be enlarged.

- Compare a wider selection of object databases and ORM tools with each other and other relational databases. Here are some examples of ORM tools, object databases and relational databases that could be included:

  - Object databases: Objectivity [24], ObjectStore [25], Ozone [4], Gemstone Facets [71], etc.
  - ORM tools: Toplink [27], Apache OJB [17], etc.
  - Relational databases: MySQL [23], Oracle [26], etc.

- Currently JPOX [9] provides a JDO and JPA implementation for db4o. Versant already support JDO. Create a JDO OO7 implementation switching between Versant and db4o to find differences in performance when using the JDO interface.

- Investigate the relational TPC database benchmarks [28]: TPC-C and the newly created TPC-E benchmarks. Investigate if these benchmarks can be used with object databases through the use of JDO and JPA standards. If they can, compare TPC benchmark results with that of OO7.

Bibliography

[1] Ergonomics in the 5.0 Java Virtual Machine. http://java.sun.com/docs/hotspot/gc5.0/ergo5.html.

[2] Java Virtual Machine. http://java.sun.com/j2se/1.5.0/docs/guide/vm/index.html.

[3] William R. Cook discussion on Query 8. http://developer.db4o.com/forums/thread/29669.aspx.

[4] OZONE. http://www.ozone-db.org, 2006.

[5] Persistence. http://en.wikipedia.org/wiki/, 2007.

[6] Xdoclet. http://xdoclet.sourceforge.net/xdoclet/index.html, 2007.

[7] Ant. http://ant.apache.org/, 2008.

[8] Hibernate. http://www.hibernate.org/, 2008.

[9] JPOX. http://www.jpox.org/, 2008.

[10] LINQ. http://en.wikipedia.org/wiki/Linq, 2008.

[11] The Object Management Group (OMG). http://www.omg.org/, 2008.

[12] ODBMS.ORG. http://www.odbms.org/, 2008.

[13] Open Source Initiative. http://www.opensource.org/, 2008.

[14] PostgreSQL. http://www.postgresql.org/, 2008.

[15] Versant. www.versant.com, 2008.

[16] Your Dictionary. http://www.yourdictionary.com/empirical, 2008.

[17] Apache Object Relational Bridge (OBJ). http://db.apache.org/ojb/, 2009.

[18] Embedded Database. http://en.wikipedia.org/wiki, 2009.

[19] EXT3 file system. http://en.wikipedia.org/wiki/Ext3, 2009.

[20] Hsqldb. http://hsqldb.org/, 2009.

[21] Jasper Reports. http://jasperforge.org/plugins/, 2009.

[22] Java Platform Standard Edition, 2009.

[23] MySQL. http://www.mysql.com/, 2009.

[24] Objectivity. http://www.objectivity.com/, 2009.

[25] ObjectStore. http://www.progress.com/objectstore/index.ssp, 2009.

[26] Oracle. http://www.oracle.com/index.html, 2009.

[27] Oracle TopLink. http://www.oracle.com/technology/products/ias/toplink/index.html, 2009.

[28] Transaction Processing Performance Council (TPC). www.tpc.org, 2009.

[29] X Window Systems. http://en.wikipedia.org/wiki/XWindow, 2009.

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

[31] S.W Ambler. Mapping Objects to Relational Databases: O/R Mapping In Detail. www.ambysoft.com/mappingObjectsTut.html, 2006.

[32] T. Anderson, A. Berre, M. Mallison, H. Porter, and B Schneider. The HyperModel benchmark. In F. Bancilhon, C. Thanos, and D. Tsichritzis, editors, *The HyperModel benchmark, In Proceedings Conference on Extending Database Technology*, volume Springer-Verlag Lecture Notes, pages 416, 317–331., Venice, Italy, March 1990 1990. Springer-Verlag.

[33] M. P. Atkinson, L. Daynès, M. J. Jordan, T. Printezis, and S. Spence. An orthogonally persistent Java. *SIGMOD Rec.*, 25(4):68–75, 1996.

[34] Malcolm Atkinson and Mick Jordan. First International Workshop on Persistence and Java. Technical report, Mountain View, CA, USA, 1996.

[35] Malcolm Atkinson and Mick Jordan. Proceedings of the Second International Workshop on Persistence and Java. Technical report, Mountain View, CA, USA, 1997.

[36] Malcolm Atkinson and Mick Jordan. A Review of the Rationale and Architectures of PJama: a Durable, Flexible, Evolvable and Scalable Orthogonally Persistent Programming Platform. Technical report, Mountain View, CA, USA, 2000.

[37] Barry and Associates. Database Concepts and Standards. http://www.service-architecture.com/database/articles/index.html, 2004.

[38] Douglas K. Barry. *The object database handbook*. John Wiley & Sons, Inc., 1996.

[39] Christian Bauer and Gavin King. *Hibernate in Action*. Manning Publications Co., Greenwich, CT, USA, 2005.

[40] C.J Berg. *Advanced Java Development for Enterprise Applications*. New Jersey, Second edition, 2000.

[41] E. Bertino and L. Martino. *Object-Oriented Database Systems*. Addison-Wesley, Reading MA., 1993.

[42] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. *SIGPLAN Not.*, 41(10):169–190, 2006.

[43] Don Box and Anders Hejlsberg. LINQ: .NET Language-Integrated Query. http://msdn.microsoft.com/en-us/library/bb308959.aspx, February 2007.

[44] Alan W. Brown. *Object-Oriented Databases and Their Applications to Software Engineering*. McGraw-Hill, Inc., New York, NY, USA, 1991.

[45] M. J. Carey, D.J Dewitt, and J. F. Naughton. The OO7 benchmark. CS Tech Report. Technical report, University of Wisconsin-Madison, 1993.

[46] M. J. Carey, D.J Dewitt, and J. F. Naughton. The OO7 benchmark. CS Tech Report. Technical report, University of Wisconsin-Madison, January 1994.

[47] Michael J. Carey, David J. DeWitt, Chander Kant, and Jeffrey F. Naughton. A status report on the OO7 OODBMS benchmarking effort. *SIGPLAN Not.*, 29(10):414–426, 1994.

[48] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 Benchmark. *SIGMOD Rec.*, 22(2):12–21, 1993.

[49] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. Interpreting OO7 Results. http://ftp.cs.wisc.edu/007/interpreting.oo7, January 1994.

[50] Michael J. Carey, David J. DeWitt, Jeffrey F. Naughton, Mohammad Asgarian, Paul Brown, Johannes E. Gehrke, and Dhaval N. Shah. The BUCKY object-relational benchmark. *SIGMOD Rec.*, 26(2):135–146, 1997.

[51] R.G.G Cattel, D. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez. *The Object Data Standard: ODMG 3.0.* Morgan Kaufmann Publishers, 2000.

[52] R. G. G. Cattell and J. Skeen. Object operations benchmark. *ACM Trans. Database Syst.*, 17(1):1–31, 1992.

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

[54] Akmal B. Chaudhri. An annotated bibliography of benchmarks for object databases. *SIGMOD Rec.*, 24(1):50–57, 1995.

[55] Akmal B. Chaudhri and Mary Loomis. *Object databases in practice.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[56] Akmal B. Chaudhri and Norman Revell. Benchmarking Object Databases: Past, Present & Future. http://citeseer.ist.psu.edu/rd/0qwww.cs.city.ac.ukqSqMay 1994.

[57] Akmal B. Chaudhri and Roberto Zicari. *Succeeding with object databases.* John Wiley & Sons, Inc., New York, NY, USA, 2001.

[58] Vincent Coetzee and Robert Walker. Experiences using an ODBMS for a high-volume internet banking system. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 334–338, New York, NY, USA, 2003. ACM.

[59] William R. Cook and Carl Rosenberger. Native Queries for Persistent Objects, A Design White Paper. *Dr. Dobb's Journal*, 2006.

[60] Laurent Daynès and Grzegorz Czajkowski. High-Performance, Space-Efficient, Automated Object Locking. In *Proceedings of the 17th International Conference on Data Engineering*, pages 163–172, Washington, DC, USA, 2001. IEEE Computer Society.

[61] db4objects. *db4o Tutorial*, 7.0 edition, 2007.

[62] db4objects. *Reference Documentation for db4o Java*, 7.3 edition, April 2008.

[63] db4objects Inc. db4o. http://db4o.com, 2008.

[64] db4objects Inc. Bug in db4o related to cascadeOnDelete. http://tracker.db4o.com/browse/COR-1008, 2009.

[65] C. Delobel, C Lecluse, and P. Richard. *Databases: From Relational to Object-Oriented Systems.* International Thomson Publishing, London, 1995.

[66] David J. DeWitt, Philippe Futtersack, David Maier, and Fernando Vélez. A Study of Three Alternative Workstation-Server Architectures for Object Oriented Database Systems. In *VLDB '90: Proceedings of the 16th International Conference on Very Large Data Bases*, pages 107–121, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[67] David J. DeWitt, Jeffrey F. Naughton, John C. Shafer, and Shivakumar Venkataraman. Parallelizing OODBMS traversals: a performance evaluation. *The VLDB Journal*, 5(1):003–018, 1996.

[68] EJB3 Expert Group. *JSR 220: Enterprise JavaBeans Version 3.0 Java Persistence API.* Sun Microsystems, Santa Clara, CA,, June 2005.

[69] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems.* The Benjamin/Cummings Publishing Company Inc, second edition, 1994.

[70] Martin Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley, 2003.

[71] GemStone Systems. Gemstone Facets. http://www.facetsodb.com.

[72] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. STMBench7: a benchmark for software transactional memory. *SIGOPS Oper. Syst. Rev.*, 41(3):315–324, 2007.

[73] Zhen He, Stephen Blackburn, Luke Kirby, and John N. Zigman. Platypus: Design and Implementation of a Flexible High Performance Object Store. In *POS-9: Revised Papers from the 9th International Workshop on Persistent Object Systems*, pages 100–124, London, UK, 2001. Springer-Verlag.

[74] Hibernate. *Hibernate Annotations Reference Guide*, 3.2.1.GA edition, 2007.

[75] Hibernate. *Hibernate Tools Reference Guide*, 3.2.0.beta10 edition, 2007.

[76] Hibernate. *Hibernate Reference Manual*, 3.2.2 edition, 2008.

[77] A. Ibrahim and W. Cook. OO7. http://sourceforge.net/projects/oo7/.

[78] A. Ibrahim and W. Cook. Automatic Prefetching by Traversal Profiling in Object Persistence Architectures. In *Proc. of the European Conference on Object-Oriented Programming (ECOOP)*, volume 4067/2006 of *Lecture Notes in Computer Science*, pages 50–73. Springer Berlin / Heidelberg, 2006.

[79] Java Docs API. References in Java. http://java.sun.com/javase/6/docs/api/java/lang/ref/package-summary.html.

[80] M Jordan. A Comparative Study of Persistence Mechanisms for the Java Platform. Technical report, September 2004.

[81] Sheetal Vinod Kakkad. *Address translation and storage management for persistent object stores.* PhD thesis, 1997. Adviser-Paul R. Wilson.

[82] Alfons Kemper and Guido Moerkotte. *Object-Oriented Database Management: Applications in Engineering and Computer Science.* Prentice-Hall Inc., New Jersey, 1994.

[83] M.A. Ketabchi, T. Risch, S. Mathur, and J. Chen. Comparative Analysis of RDBMS and ODBMS: A Case Study. pages 528–535. Compcon Spring '90. Intellectual Leverage. Digest of Papers. Thirty-Fifth IEEE Computer Society International Conference., Feb.-Mar 1990.

[84] Won Kim. *Introduction to Object-Oriented Databases*. The MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusetts 02142, 1990.

[85] P. Kroha and F.H Lochovsky, editors. *Object-Oriented Concepts, Databases, and Applications*. Addison-Wesley, Reading MA, 1989.

[86] P. Kruchten. Architectural Blueprint - The 4+1 View Model of Software Architecture. *IEEE Software*, 12(6):42–50, November 1995.

[87] Ying Guang Li, Stéphane Bressan, Gillian Dobbie, Zoé Lacroix, Mong Li Lee, Ullas Nambiar, and Bimlesh Wadhwa. XOO7: applying OO7 benchmark to XML query processing tool. In *CIKM '01: Proceedings of the tenth international conference on Information and knowledge management*, pages 167–174, New York, NY, USA, 2001. ACM.

[88] Diego R. Llanos. TPCC-UVa: an open-source TPC-C implementation for global performance measurement of computer systems. *SIGMOD Rec.*, 35(4):6–15, 2006.

[89] M.E.S Loomis. *Object Databases: The Essentials*. Addison-Wesley, Reading MA., 1995.

[90] Alonso Marquez, Stephen Blackburn, Gavin Mercer, and John N. Zigman. Implementing Orthogonally Persistent Java. In *POS-9: Revised Papers from the 9th International Workshop on Persistent Object Systems*, pages 247–261, London, UK, 2001. Springer-Verlag.

[91] Jeffrey C. Mogul. Brittle Metrics in Operating Systems Research. In *HOTOS '99: Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems*, pages 90–96, Washington, DC, USA, 1999. IEEE Computer Society.

[92] Jim Paterson, Stefan Edlich, Henrik Hörning, and Reidar Hörning. *The Definitive Guide to db4o*. Apress, Berkely, CA, USA, 2006.

[93] Craig Russell. *Java Data Objects JSR 12*. Sun Microsystems Inc., version 1.0.1 edition, May 2004.

[94] Sun Microsystems Inc. J2SE 5.0 Performance White Paper. java.sun.com/performance/reference/whitepapers.

[95] A. Tiwary, V. Narasayya, and H. Levy. Evaluation of OO7 as a system and an application benchmark. In OOPSLA Workshop on Object Database: Behavior, Benchmarks, and Performance. citeseer.ist.psu.edu/tiwary95evaluation.html, 1995.

[96] Pieter van Zyl, Derrick G. Kourie, and Andrew Boake. Comparing the performance of object databases and ORM tools. In *SAICSIT '06: Proceedings of the 2006 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, pages 1–11, Republic of South Africa, 2006. South African Institute for Computer Scientists and Information Technologists.

[97] Pieter van Zyl, Derrick G. Kourie, and Andrew Boake. Erratum in Comparing the Performance of Object Databases and ORM Tools. http://www.odbms.org/download/027.02August 2007.

[98] Versant. *Java VERSANT Interface Usage Manual.* Versant Corporation, release 7.0.1.3 edition, 2008.

[99] Versant. *Versant JDO Interface User's Guide.* Versant, 2008.

[100] Versant Corporation. *Versant Database Fundamentals Manual*, release 7.0.1.4 edition, 2008.

[101] Adam Welc, Suresh Jagannathan, and Antony Hosking. Safe futures for Java. *SIGPLAN Not.*, 40(10):439–453, 2005.

[102] Benjamin G. Zorn and Akmal B. Chaudhri. Object Database Behavior, Benchmarks, And Performance: Workshop Addendum. In *OOPSLA '95: Addendum to the proceedings of the 10th annual conference on Object-oriented programming systems, languages, and applications (Addendum)*, pages 159–163, New York, NY, USA, 1995. ACM.

[103] John Zukowski. Generating Integer Random Numbers. http://java.sun.com/developer/JDCTechTips/2001/tt0925.html, September 2001.

Appendix A

Appendix

A.1 How to obtain the code and contribute

This study supports the OSS ideal and would like to share the code with other researchers and collaborators. The code will be available as a project on Sourgeforge: http://sourceforge.net/projects/oo7j. Note that the code is distributed under the GNU General Public License (GPL).

The original C++ OO7 implementation can be found here:
http://ftp.cs.wisc.edu/OO7/.

A website and wiki will be created where users and vendors can leave comments and ideas on how to improve their configurations using OO7.

They will be allowed to take the code, add improved configurations and submit them for review and inclusion into the main branch of the project. All additions and configurations must be clearly explained and motivated.

A.2 OO7 benchmark operations

The following table contains a summary of the OO7 benchmark operations. This data has been obtained from Carey et al. [46, 48]

Operation Name	Description of operation

Operation Name	Description of operation
Traversal T1:Raw traversal speed	tTraverse the assembly hierarchy. As each base assembly is visited, visit each of its referenced unshared composite parts. As each composite part is visited, perform a depth first search on its graph of atomic parts. Return a count of the number of atomic parts visited when done.
Traversal T2 (a,b and c): Traversal with updates	Repeat Traversal #1, but update objects during the traversal. There are three types of update patterns in this traversal. In each, a single update to an atomic part consists of swapping its (z, y) attributes The three types of updates are: (a) Update one atomic part per composite part. (b) Update every atomic part as it is encountered. (c) Update each atomic part in a composite part four times. When done, return the number of update operations that were actually performed.
Traversal T3 (a,b and c): Traversal with indexed updates	Repeat Traversal T2, except that now the update is on the date field, which is indexed. The specific update is to increment the date if it is odd, and decrement the date if it is even.
Traversal T6: Sparse traversal speed.	Traverse the assembly hierarchy. As each base assembly is visited, visit each of its referenced unshared composite parts. As each composite part is visited, visit the root atomic part. Return a count of the number of atomic parts visited when done.

Operation Name	Description of operation
Traversals T8 and T9: Operations on Manual.	Traversal T8 scans the manual object, counting the number of occurrences of the character \I." Traversal T9 checks to see if the first and last character in the manual object are the same.
Query Q1: exact match lookup	Generate 10 random atomic part id's; for each part id generated, lookup the atomic part with that id. Return the number of atomic parts processed when done.
Queries Q2, Q3, and Q7.("These queries are most interesting when considered together"[46])	Query Q2: Choose a range for dates that will contain the last 1% of the dates found in the database's atomic parts. Retrieve the atomic parts that satisfy this range predicate.
	Query Q3: Choose a range for dates that will contain the last 10% of the dates found in the database's atomic parts. Retrieve the atomic parts that satisfy this range predicate.
	Query Q7: Scan all atomic parts.
Query Q4: path lookup	Generate 100 random document titles. For each title generated, find all base assemblies that use the composite part corresponding to the document. Also, count the total number of base assemblies that qualify.
Query Q5: single-level make	Find all base assemblies that use a composite part with a build date later than the build date of the base assembly. Also, report the number of qualifying base assemblies found.

Operation Name	Description of operation
Query Q8: ad-hoc join	Find all pairs of documents and atomic parts where the document id in the atomic part matches the id of the document. Also, return a count of the number of such pairs encountered.
Structural Modification: Insert	Create five new composite parts, which includes creating a number of new atomic parts (100 in the small configuration, 1000 in the large, and five new document objects) and insert them into the database by installing references to these composite parts into 10 randomly chosen base assembly objects.
Structural Modification 2: Delete	Delete the five newly created composite parts (and all of their associated atomic parts and document objects).

A.3 Comparisons results that include db4o in embedded mode

In this section db4o embedded mode is compared with Hibernate and Versant which run in client-server mode. This is not a fair comparison. It is included because db4o recommended running db4o in embedded mode. These results might be of interest if one need to decided between using an embedded database or a client/server database.

Only the results are provided. The discussion of these results fall outside the scope of the current study.

A.3.1 Creation

The embedded db4o lazy configuration has a dramatic improvement from 7700 seconds to 15 seconds.

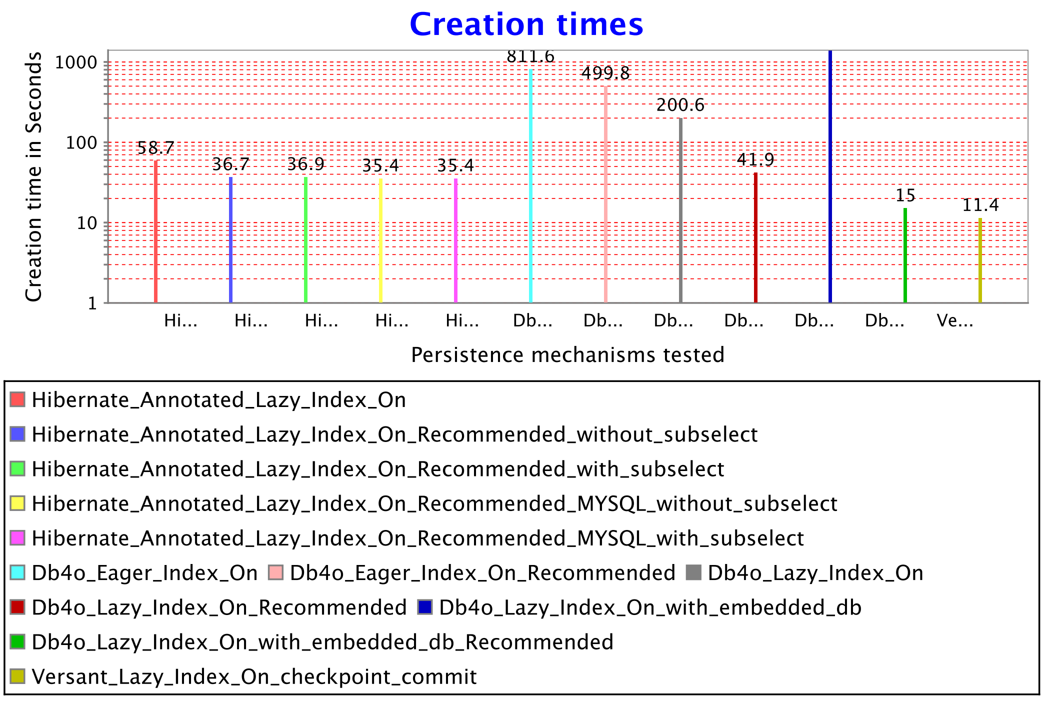


Figure A.1: Creation times

A.3.2 Traversals

The embedded db4o configuration using the recommendations are the *fastest db4o* configuration.

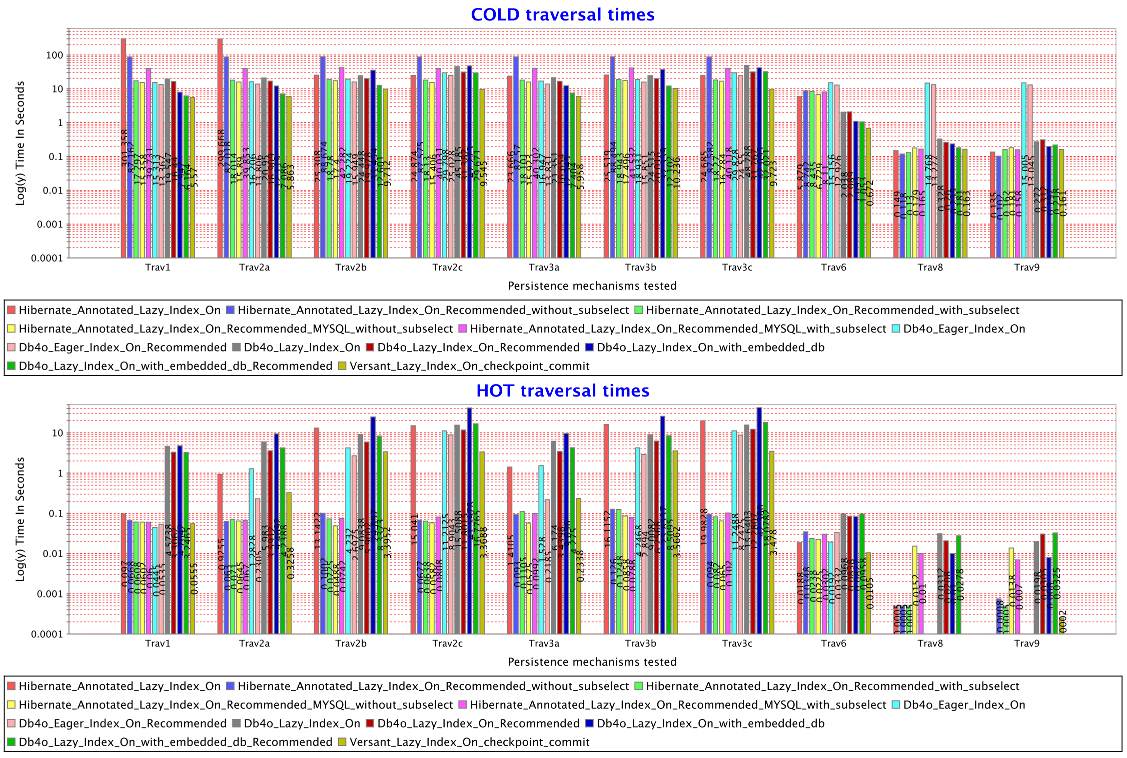
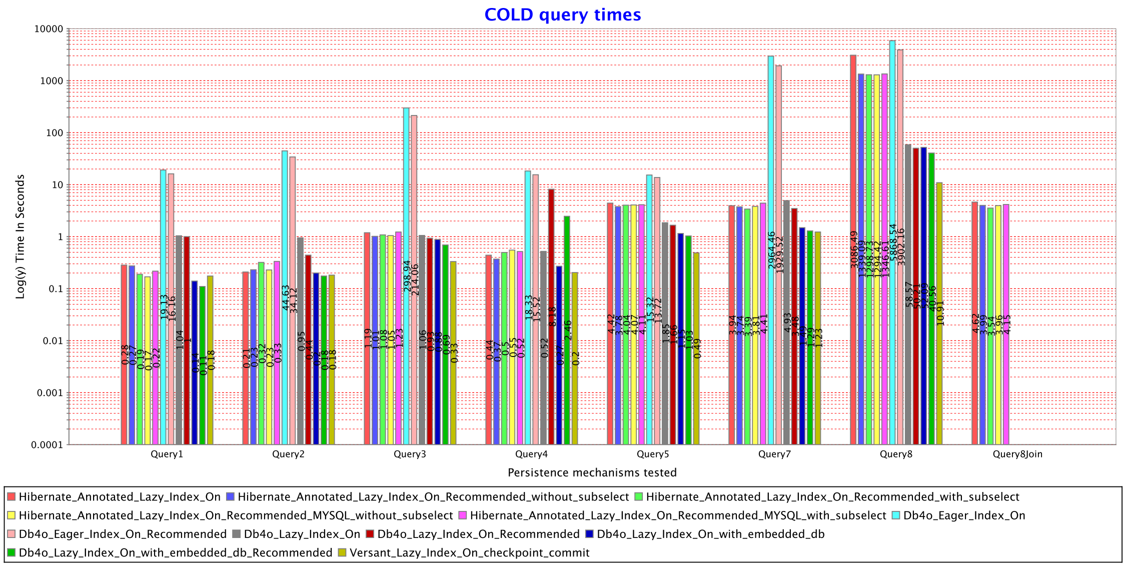


Figure A.2: Comparison results of traversals

A.3.3 Queries

For the query runs the client-server db4o configurations are faster in most cases than the embedded db4o configuration.



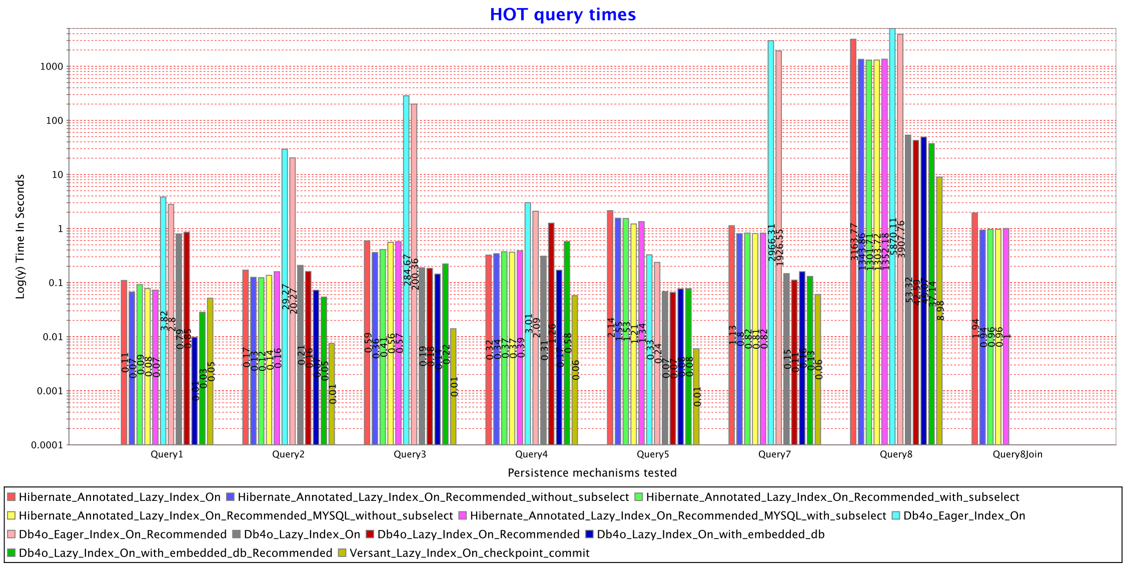
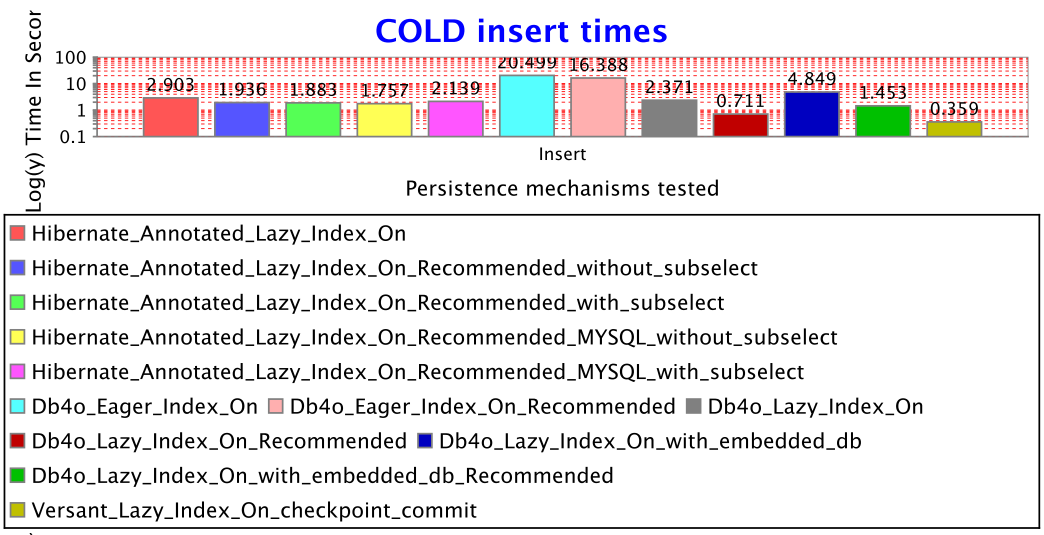


Figure A.3: Comparison results of queries

A.3.4 Modifications: Insert and delete

The embedded db4o configuration using the recommendations has now gone down from ±5 seconds down to ±1.4 seconds for cold inserts. This is a 70% improvement.



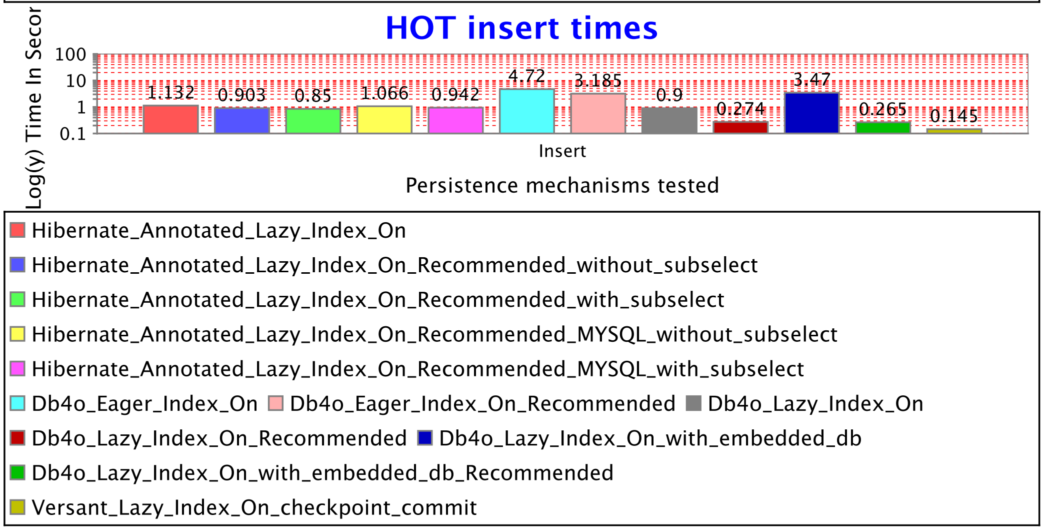
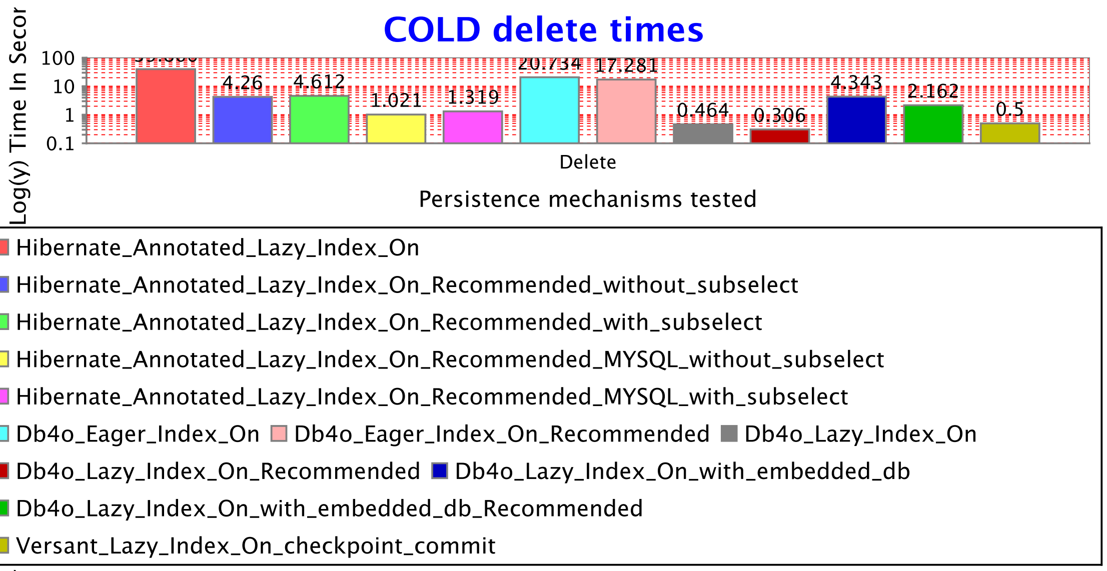


Figure A.4: Comparison result of inserts



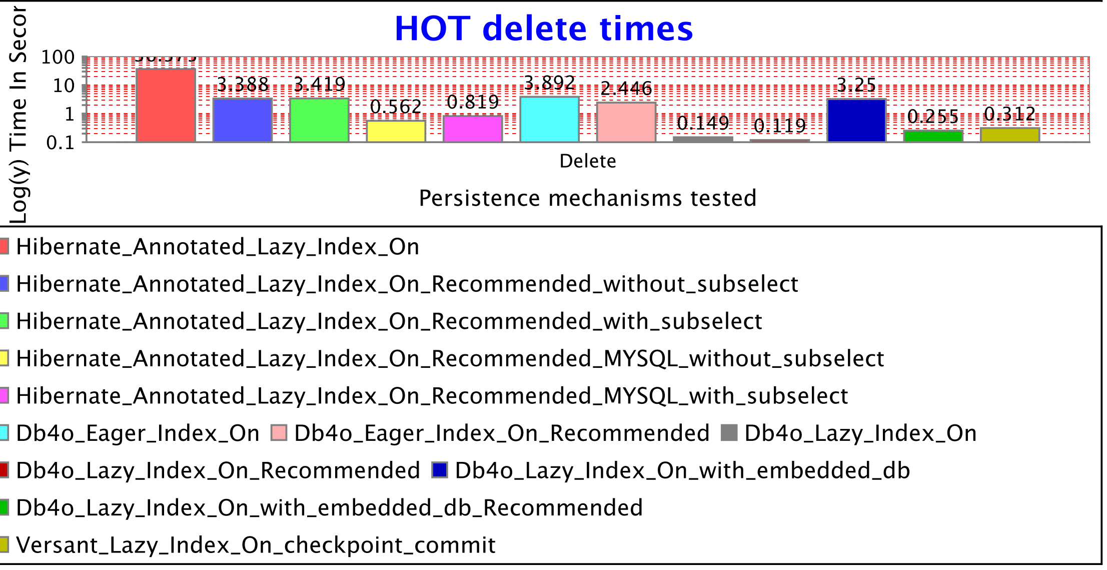


Figure A.5: Comparison result of deletes